graphic art
on the commodore 64

techniques for high resolution graphics

boris allan

First published 1983 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street,
London WC2R 3LD

ISBN 0 946408 15 7

Cover design by Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset and printed in England by Commercial Colour Press, London E7.

CONTENTS

Introduction

When I came to write a book about high resolution graphics on the Commodore 64 (the C64) I knew there were problems.

The first problem is with the manual that accompanies the machine on purchase. Commodore claim that the guide is designed to give you, the user, all the information you need to set up your equipment properly, and give you a simple fun start to computing.

Basically the user manual gives very little away about the C64, and does not really allow you to investigate some of the more powerful features of the machine. In writing this book I have assumed that you have no more than the *Commodore 64 MicroComputer User Manual* (as it says on the cover).

Here you are with a machine which has sprites, memory management, sound synthesis, and a rather simple manual as an introduction to all these features. Sound effects rate a chapter in the manual, as do sprites. Memory management is not mentioned, however, and the facility to use bit mapped graphics appears nowhere.

The second problem comes when you buy (beg, borrow, or steal) the *Programmer's Reference Guide*. To fight your way through the reference guide takes some doing. To buy it at all is a definite sign of high income. Though I refer to the reference guide in my appendices, the references are only to supplement my treatment or, sometimes, to explain what the guide really means or implies.

The third problem is how to organize my treatment. The BASIC on the C64 might politely be called rudimentary (with the emphasis on rude). As the BASIC is so rudimentary, therefore, you have often to go outside BASIC to do even simple things: such as draw a straight line.

I have provided very easy-to-use routines for high resolution graphics on the C64, and I do not want that simplicity ruined by having to explain in detail how they work. However, you — as the user — need to know how and why the routines work, just as you also need to know something about the general structure of the C64.

To program in high resolution graphics on the C64 is to actively indulge in extensive memory management, and complex manipulations of registers. To use my routines you do not require any of this knowledge, and you can treat them purely as what they are: a few (very few) routines which will allow you almost complete flexibility in the graphic art.

I have tried to resolve the third problem in this way. The discussion of

turtle graphics is as untechnical as any discussion of programming can ever be. In my chapters I am talking mainly about programming, and assuming the implementation of the turtle graphics as given.

The development and reasoning behind my implementation of high resolution graphics (used by the turtle graphics) is covered in my appendices. There are ten appendices, covering topics ranging from binary logic to character generation to memory management.

In the appendices you will find much information which is not readily accessible elsewhere. For example, it might be in the reference guide, but: one, can you afford it; two, can you understand it; and, three, can you find it? I have provided many short utility routines in the appendices, to cover some of the small but fiddling tasks that beset the programmer.

The main aim of my chapters and appendices is to help you realize how the C64 works, why it is such an amazing machine — and why it is so surprising that Commodore have not provided a better standard BASIC. The discussions in the appendices will provide you with an array of useful reference material — well, you always did want to know why some of those POKEs work like they do, didn't you?

SECTION 1

Turtle Graphics

CHAPTER 1
The Idea of Turtle Graphics

Suppose you are being asked directions in a street. You are more than likely to say so far forward, turn in this direction, forward a bit, turn, and so on.

Think, then, how you draw a square on a piece of paper. You draw a line forward a certain amount, then draw a line at right angles to the first line, continue for the same distance as before, turn through a right angle...

Suppose you have a piece of graph paper (also very useful for user defined characters), and are asked to draw a square. Let us further suppose that you use the squares on the graph paper to act as your guides for the drawing. You use 'coordinates' on the graph paper to ease the drawing.

You are asked next to draw a square at 30 degrees to the existing square (on the graph paper) without use of a protractor. You have to calculate the correct positions of the corners by use of trigonometry. Not so simple.

If you had a protractor, you could measure 30 degrees and mark off the length of a side, and so forth for the other sides. Doing geometry by moves forward and turns seems far simpler: assuming we have the technology (eg the protractor).

Intrinsic geometries

What we have been examining are the intrinsic features of a square. A square has four sides and four right angles, and these are intrinsic features of a square — shared by all squares.

Any actual square has certain extrinsic features — it is here or there, it is this size, it is tilted at this angle. These are all characteristics of that square, not squares in general.

In a book entitled *Mindstorms*, Seymour Papert gives a short BASIC program to draw a house: it would not work on the C64 because you cannot draw lines (yet). The program works on the use of coordinates and, even for a simple little program, quite a good deal of work goes into plotting the coordinates.

If you have a roof at an angle of 60 degrees to the horizontal, try to work out the coordinates of the apex of this roof.

Papert notes also that this is not a suitable method for drawing a house because of the work involved: 'This demand however would be less serious if the program, once written, could become a powerful tool for other

projects... the BASIC program allows one particular house to be drawn in one position. In order to make a BASIC program that will draw houses in many positions, it is necessary to use algebraic variables...'

In other words, by using coordinates to draw the house, the extrinsic aspects of the house are accentuated: the only useful results might be any intrinsic aspects still in the program. This is quite a common problem in other aspects of programming.

We write a program to calculate the value of a function

```
X = .123456*Y^2.3
```

where the .123456 and 2.3 are specific to just one problem. It is good programming to generalize somewhat (accentuate the intrinsic features) and use

```
X = C1*Y^E1
```

where C1 and E1 can be supplied at the time the program is run.

A house (in the abstract terms defined by Seymour Papert) is no more than a square with a triangle on top. The intrinsic definition of a house, therefore, is

```
HOUSE: SQUARE with TRIANGLE above
```

where we would have to explain what was meant by above. We can define a

```
SQUARE : 4 LINEs with a TURN of 90 degrees at end
```

and a

```
TRIANGLE : 3 LINEs with a TURN of 120 degrees at end
```

We could then move to giving a meaning to our primitive notions, LINE and TURN. Turtle geometry is based on ideas just like these.

Turtle Geometry

This form of analysis, which was designed to accentuate the intrinsic elements of a problem, has been implemented as a programming language called LOGO. An important part of LOGO is a very flexible and easy-to-use system of high resolution graphics called Turtle Graphics.

Papert's book, *Mindstorms*, is the best way into the whole field of living and learning based on the intrinsic approach. The use of turtle graphics with children in primary schools has been extremely successful, so successful that some people think that turtle graphics is for kids.

This is wrong. One of the most popular implementations of Pascal (not a kiddy language) on microcomputers has a version of turtle graphics. Called UCSD Pascal, it was used originally for teaching students at the University

of California, San Diego. The students enjoyed using turtle graphics, as did young children, but they also learnt many complex ideas in a not-so-complex way.

There is a book by Harold Abelson and Andrea diSessa (called *Turtle Geometry*) which is the basis for an undergraduate course in mathematics. Abelson and diSessa teach at the Massachusetts Institute of Technology — and that is no kindergarten.

The scope of the *Turtle Geometry* book is vast (and it costs more than the C64 reference guide), from the very simple to the complex. The simple includes drawing a square, and the complex includes the phenomena of curved time and space. We are only going to go a little further than the square.

In turtle graphics you have control of an imaginary (and, in my version for the C64, invisible) creature called a turtle. It is called a turtle because that is the American for tortoise, and one of the first robots to move on the floor was called a tortoise (invented by Gray Walter).

The turtle responds to a very limited set of commands: in my version it either moves in a straight line, or it turns. The turtle drags a pen behind it, and when the pen is lowered a line is drawn, and when it is up it does not draw the line. My version allows plot or not plot, which is effectively the same.

The squares

My **Figure 1.1** is supposed to show a square, but the square has been compressed by the screen dump. However, the square was drawn by use of the following lines of program:

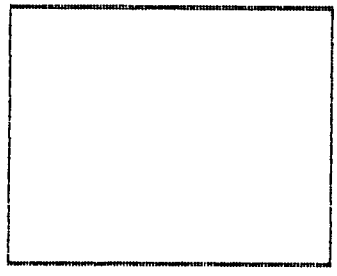
```
10 INPUT DI
20 GOSUB 20000
30 FOR KK = 1 TO 4
40 GOSUB 21000
50 AN = AN + 90
60 NEXT KK
70 GOTO 70
```

I will explain about this program, without actually giving the content of the subroutines called.

The first line, line 10, inputs the size of the square. Any distance we are going to move our turtle is always called DI in this version (DIstance). Line 20000 sets up the high resolution graphics system, and initializes the turtle graphics variables.

The subroutine 21000 is called 4 times, and is the line drawing routine. 21000 will draw a line a distance DI forward, in the direction in which the turtle is facing. The turtle faces in a direction of AN degrees, where up is 0, and the angle increases counterclockwise.

Figure 1.1



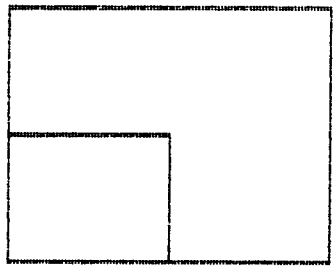
After a line has been drawn, the angle is increased by 90 degrees (equivalent to a turn of 90 degrees to the left), so that when the action is repeated the turtle moves at right angles to its present track. Line 70 merely keeps the high resolution display on the screen.

Figure 1.2 was taken from the result of this program:

```
10 INPUT DI
20 GOSUB 20000
25 FOR KI = 1 TO 2
30 FOR KK = 1 TO 4
40 GOSUB 21000
50 AN = AN + 90
60 NEXT KK
65 DI = DI/2 : NEXT KI
70 GOTO 70
```

The difference, you notice, is at lines 25 and 65. We can think of the section from 30 to 60 as an intrinsic definition of how a square is drawn. Every time we use the routine, with a different DI, we draw a different size of square. The side of the square is equal to DI.

Figure 1.2



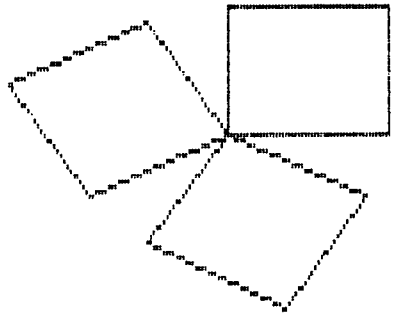
The routine is used twice (cf line 25) and before leaving the loop the size DI is halved. There are only two occasions on which the square is drawn: the first with side of DI, and the second with side of DI/2.

Figure 1.3 shows three squares, tilted at various angles. The program for this is somewhat like the last:

```
10 INPUT DI,N1
20 GOSUB 20000
25 FOR KI = 1 TO N1
30 FOR KK = 1 TO 4
40 GOSUB 21000
50 AN = AN + 90
60 NEXT KK
65 AN = AN + 360/N1 : NEXT KI
70 GOTO70
```

with the difference being that the size of the side is not altered, but the angle at which the turtle starts to draw the square alters each time. For Figure 1.3, the value N1 to be input is 3, and the angle to be turned is 360/3 = 120 degrees.

Figure 1.3



There are five squares tilted in **Figure 1.4**, and it is around this time that the C64's lack of speed becomes rather obvious. For some variety, **Figures 1.5** and **1.6** also show tilted squares.

The first thing to notice is the ease with which we made the transition from a simple square to many squares. You still do not know what the routines at 20000 and 21000 actually do, but I hope you will appreciate that the extension is rather simple.

Figure 1.4

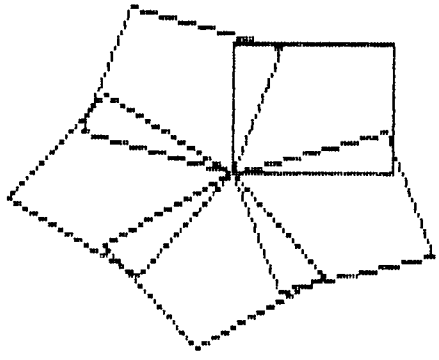


Figure 1.5

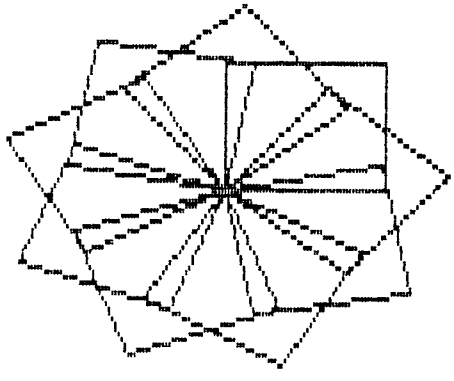
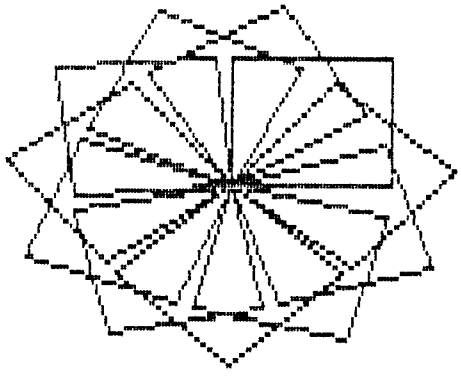


Figure 1.6



The second thing is that I could have made the square routine into a proper subroutine, but I am wary of too many nested subroutines. The third is that the turtle always faces in the direction it started in after drawing the circle — this is why AN = AN + 360/N1 worked.

The turtle routines

```
19999 REM CALL TURTLE GAPHICS SYSTEM
20000 LX=150 : LY=100 : NX=LX : NY=LY :
AN=0 : PE=1 : REM INITIALIZATIONS
20010 GOSUB 10000 : REM HIRES INITIALIZA
TIONS
20020 RETURN
```

The routine at 20000 sets up the turtle graphics system, and initializes certain variables.

LX and LY are the coordinates of the point to which the turtle last moved (and therefore the point at which any new line starts). NX and NY are the coordinates of the new point to which the turtle is to move. After the line has been drawn, we assign the values of NX and NY to LX and LY.

The angle at which the turtle is pointing is AN, and — originally — it points upwards. If the pen is down then PE = 1; if the pen is up (and no lines are to be drawn) PE = 0.

The call to the subroutine at line 10000 initializes high resolution graphics (the full details of the workings of this routine are contained in Appendix H). The listing of subroutine 10000 is given in Appendix H.

That is all there is to the initialization routine.

```
20999 REM THE TURTLE DRAWING ROUTINE
21000 NX=LX-DI*SIN(AN*PI/180) : REM NEW
COORDINATE FOR X
21010 NY=LY-DI*COS(AN*PI/180) : REM NEW
COORDINATE FOR Y
21020 IF PE=1 THEN GOSUB 12000 : REM DR
AW A LINE IF PEN IS DOWN
21030 LY=NY : LX=NX : REM REMEMBER WHER
E YOU HAVE BEEN
21040 RETURN
```

We draw our turtle lines by use of the subroutine at 21000. The routine needs the user to supply the value of DI (the distance moved) — and the values of AN and PE.

The variable PI represents the value pi (easier to show than the shift up arrow), and is used in the conversion of angle AN from degrees to radians.

We need to convert the angle because the SINe and COSine functions only operate on radians.

In lines 21000 and 21010 we calculate the new coordinates (NX and NY) from the last coordinates (LX and LY), in conjunction with the DIstance to be moved. If the pen is down we then draw a line between the two sets of coordinates, using a high resolution routine given in Appendix H (ie 12000). The values of the new coordinates are then copied into the last coordinates.

It may seem strange that there are only two routines, but they are based on the work explained in the appendices. The next chapters will show some applications.

CHAPTER 2
Investigating Spirals

We are going to base our discussion in this chapter on some results of this little program

```
10 INPUT A1,I1
20 GOSUB 20000
30 AN = AN+A1
40 DI = DI+I1
50 GOSUB 21000
60 GOTO 30
```

and so first we will explain the program.

There are two inputs A1 and I1. The turtle moves and turns according to the values of AN and DI, and the first of the two inputs effectively measures the angle through which the turtle will turn at each stage during the program.

The second input measures the increase in the distance travelled at each stage of the program. The angle turned remains constant but the distance increases at each turn and move.

The turtle graphics system is set up (line 20) and then the turtle is turned through A1 degrees. The turning is accomplished by increasing the value of AN (or by decreasing, to turn the other way). The size of the distance to be moved is also altered, and the distance increases by the value of the variable I1.

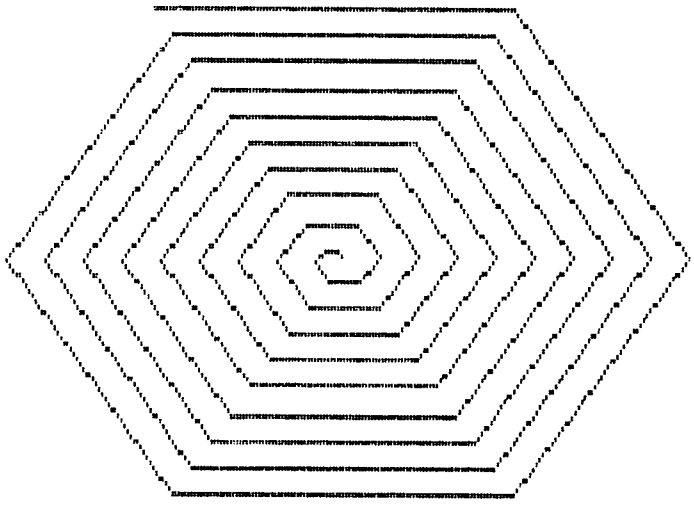
The line on the screen is plotted (at an angle of AN and for a distance DI) as a consequence of program line 50 (which calls 21000); and, at the unconditional jump GOTO 30, the sequence is repeated.

A spiral

The first outcome of the program was **Figure 2.1**, with A1 (the increase in angle) equal to 60. An ordinary hexagon can be drawn by moving a fixed distance, turning through 60 degrees, and repeating six times.

This spiral looks as if it is trying to be a hexagon, and the shape of the spiral is highly regular. 60 is a factor of 360, and thus this might have been expected.

Figure 2.1



With spirals and such like, small changes in angles can produce big changes in the effects, so we will try a small change.

Figures 2.2 and **2.3** should be seen as one. Both are examples of a kind of sixness, and both differ from 60 degrees by only one degree. Figure 2.3 has a turn of 59 degrees, and Figure 2.2 has a turn of 61 degrees. Both seem almost to contain spirals within spirals.

Consider Figure 2.3 (all the figures had to be stopped part way through). It looks as if we are looking from the top on a six-sided twisted tower. The twist seems to be counterclockwise, but it is the vertices of the spiral slowly slipping backwards from where they would have been, if the angle had been 60 degrees.

There is a clear resemblance between Figures 2.2 and 2.3, in that one seems the reverse of the other. Whereas Figure 2.3 shows slipping, Figure 2.2 shows overtaking. In Figure 2.2 we notice that the vertices are beginning to creep round in the direction of the spiral. The twist is clockwise in the case of Figure 2.2.

Because a one degree difference is so small compared to 60 degrees, the two curves are almost symmetrical. In a similar manner, if z is small enough then

$1 - z$ is almost equal to $1/(1 + z)$

eg if $z = .001$, then $1 - z = .999$, and the reciprocal of $1 + z = 1.001$ is $1/1.001 = .9990009$. That is fairly exact. If however $z = .5$, then .5 does not come close to the reciprocal of 1.5 (which is .66666667).

If the differences were large, the curves would not appear to be alike.

Figure 2.2

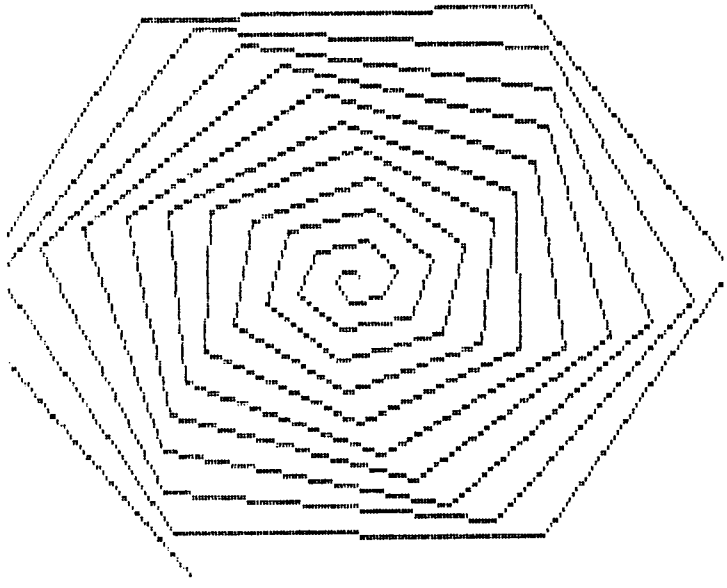
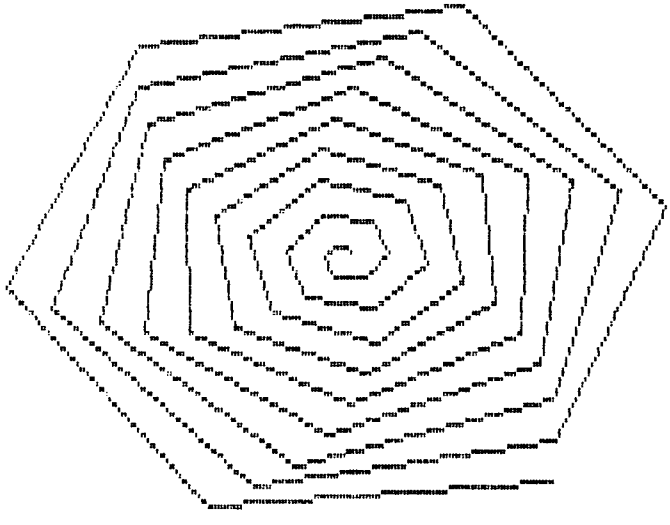


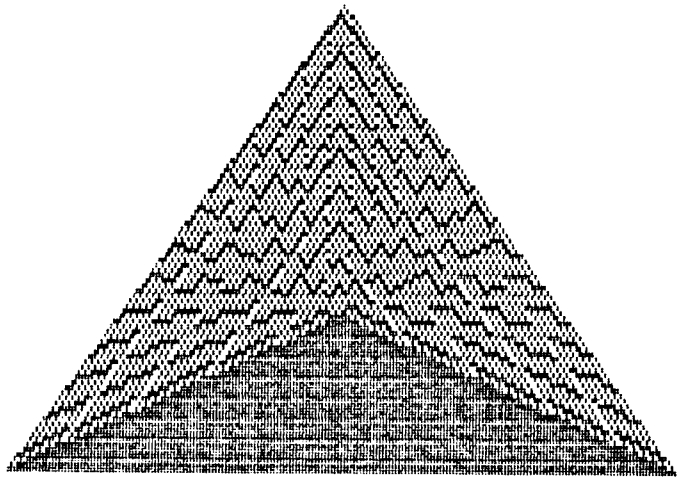
Figure 2.3



A triangle?

Using the same program, with an angle of 120 degrees, is even more interesting. Consider **Figure 2.4.**

Figure 2.4



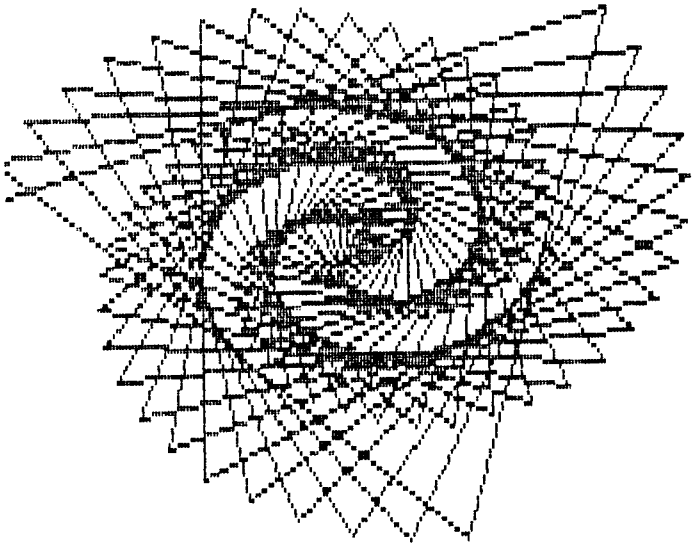
We have a triangle. The reason why this triangle is so interesting is the way in which the lines we have drawn have interacted with each other. Where there were vertical lines we have a solid block of black. Where the lines were at an angle they interfere and we have interesting patternings.

The reasons behind the patternings become clear if you read Appendix G on Block Graphics. When we use the high resolution method, we POKE values into locations, and to draw a diagonal line is not to draw a straight line. An examination of some of the tilted squares will confirm that assertion.

The interference patterns are due to the drawing of straight lines, in directions where there are no lines. The only straight lines on the screen are horizontal and vertical. Most drawn lines are thus at an angle. This is especially clear if you examine Figure 1.6. In that figure, because there are so many squares at so many differing angles, we are more likely to find non-straight lines.

Figure 2.5, to be matched with **Figure 2.8**, is an open sort of spiral, with three inner spiral arms. The three spiral arms are obviously related to the triangle shape we have at an angle of 120 degrees.

Figure 2.5



The angle for the spiral in Figure 2.5 is 123 degrees, and the inner spiral is in the same direction of rotation as that of Figure 2.2. Both these figures are cases where the angle is slightly above that of the regular spiral.

Figures 2.3 and 2.8 also spiral in the same sense (both have a deficit compared to the regular spiral). Both are reflections of the spirals at Figures 2.2 and 2.5.

The angle for **Figure 2.6** is 121 degrees, and that for **Figure 2.7** is 119 degrees. Note how one figure is almost the reflection of the other. The inner spiral for Figure 2.6 is counterclockwise, as you would expect from Figure 2.2 and 2.5.

A beautiful pair

To attempt to draw a spiral with a very small angle (almost anything less than 15 degrees) is to produce a very open spiral — so open in some cases that it never seems to turn before it is off the screen.

To turn through 0 degrees is to follow a straight line, and to turn through 180 degrees is to double back on oneself. What happens when we turn through 179 degrees? **Figure 2.9** is what happens.

Figure 2.6

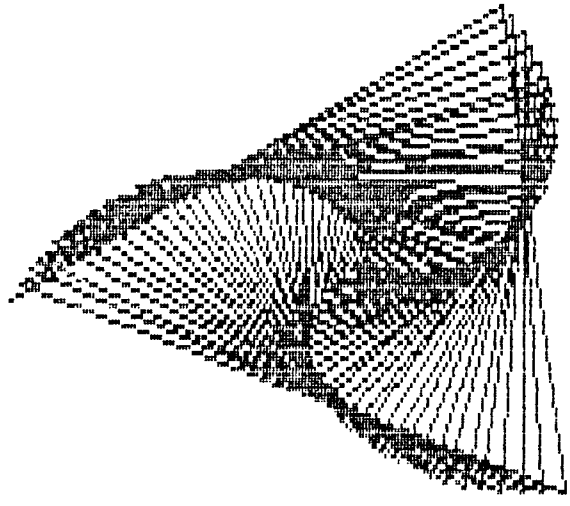


Figure 2.7

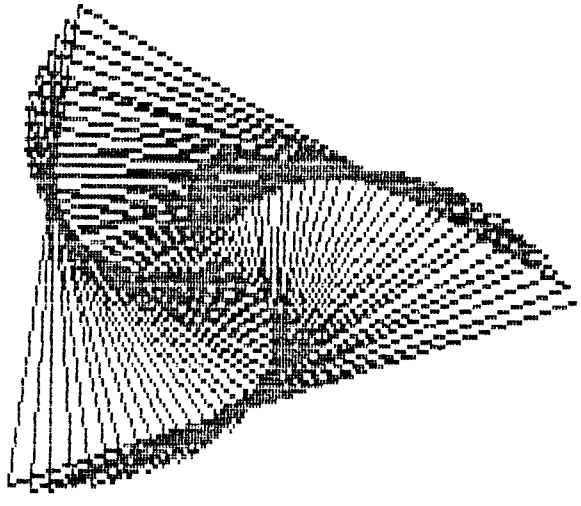


Figure 2.8

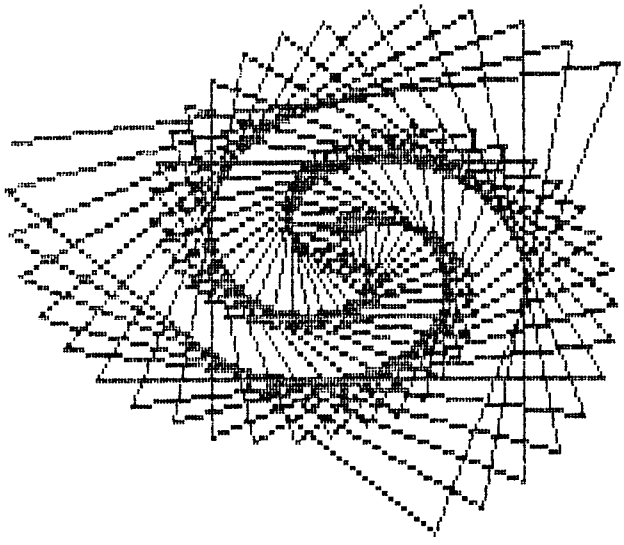
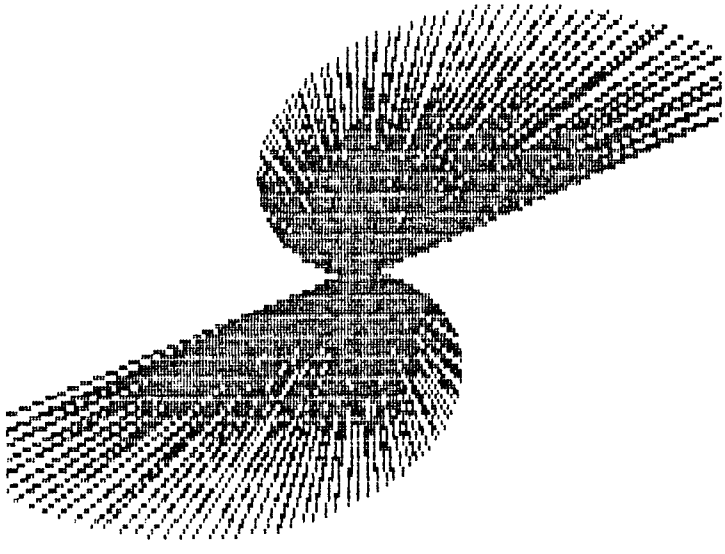
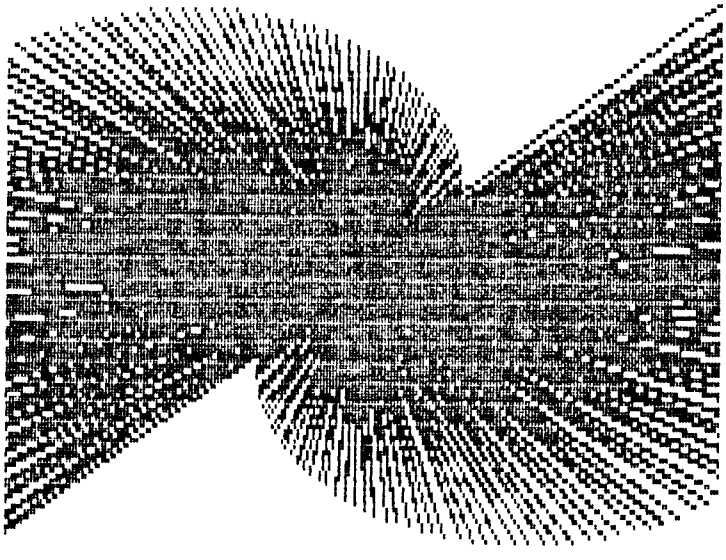


Figure 2.9



Using an angle of 181 degrees produces **Figure 2.10**. I stopped it rather later in the process to try to see how it turned out — however it took so long that boredom caught up with me first. This figure was the clearest reminder so far of just how slow the C64 high resolution graphics can be in BASIC — perhaps I should learn machine code?

Figure 2.10



The outside of the spiral as it developed seemed to follow a logarithmic spiral, such as those on shells of snails, and similar. This is a little program to investigate nature.

Pictures like these, particularly as they slowly build up on the screen, can be riveting — especially if you are trying to predict (before the event) the probable shape of the spiral. When does the three spiral become a four spiral? Can you tell?

The point behind this excursion is to show how powerful are turtle procedures, and how they can be used so easily — even in my version, using C64 BASIC.

CHAPTER 3
The Inward Spiral

Some of the following effects can take some time to produce, because of the slowness of the C64 in the production of high resolution graphics.

By now you may have worked through some of the appendices, and you should have found the high resolution routines from Appendix H. If you have not used any of the routines, you must have been captivated by the pictures. Let us recap on some of the things we have been doing.

Turtle graphics are, as you have seen, simplicity to implement on the C64, as long as you have a means to plot lines between coordinates. There is no method to draw lines provided with the standard C64 BASIC.

My implementation of high resolution graphics is examined in detail in Appendix H, but let us first consider the implications for turtle graphics. My turtle graphics are anything but interactive. Turtle graphics are best used interactively.

Irritation comes from the long clearing of the screen (you hadn't noticed?), from the less than turtle pace of the drawing (snail's pace? — snail graphics?), and the abysmal realization that the little routine you have entered has not worked — but you can't see why. The system is not ultra user friendly, though it is not difficult to use (I managed).

It is difficult to see how it could be otherwise with the C64, though a split screen might help. The way to provide a split screen is via something called the Raster Interrupt Compare Register, a trifle too complex at the moment.

When something goes wrong, I tend to leave out the turtle graphics initialization routine. I then imagine what is happening on the screen, and can see the error messages when they occur. One mistake easily made (at least, by me) is writing 2100 where it should be 21000.

The next set of examples shows the true power of the intrinsic approach. We can magnify or we can contract — have you noticed that I used an intrinsic approach to low resolution and high resolution graphics?

Other forms of spiral

We will now examine spirals of a different type. In the previous chapter the spirals all moved outwards from the centre; the spirals in this chapter are rather less constrained in their nature.

Figure 3.1

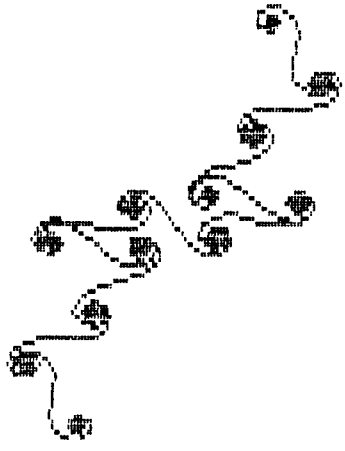
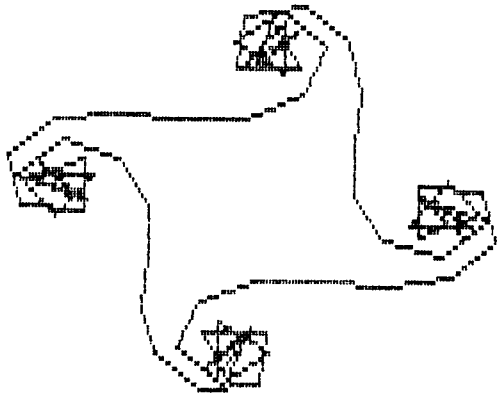


Figure 3.2



We will, again, examine the figures partly for their intrinsic interest, and also to try to illustrate another way in which we can manipulate the key variables DI and AN. Notice that we will rarely have directly to use the variables NX, NY, LX, or LY. Most of the time we do not need to know what the values are for those variables.

The program

```
10 INPUT DI,A0,A1 : AN = A0
20 GOSUB 20000
30 GOSUB 21000
40 A0 = A0 + A1
50 AN = AN + A0
60 GOTO 30
```

is called an inward spiral. The program is short but has a subtle complexity, so we will study it further.

The inputs are a distance and two angles. The first angle, A0, is the angle through which the turtle is turned (cf line 50). The angle through which the turtle is turned varies from move to move. The angle turned is increased at each move by the amount A1 (line 40).

Figure 3.1 shows an inward spiral for the starting angle (A0) of one degree, and an increment to the angle (A1) of 11 degrees. The turtle starts drawing, scuttles back and forth, and eventually ends up following the same route again. The turtle is said to follow a closed curve.

Trying with a different set of angles (5,20) produces **Figure 3.2**. As this latter figure is rather more open and observable than the previous one (though no more pleasing), I intend to investigate variants on that theme — for a while at least.

A family of spirals

All the figures from 3.2 to 3.5 are variants on one theme. The theme is A1 equal to 20, with varying values of A0 (and, of course, varying values of DI — but that is unimportant). The values for A0 are (in succession) 5, 0, 6, and 2.

Figures 3.2, and **3.4**, and **3.5** share similarities, in that the rosettes at the corners of the figures have a similar appearance. The nature of the rosettes is not too clear because of the scale of the figures.

Though all look similar, they are all, in fact, very different: Figure 3.2 has four corners, whereas Figures 3.4 and 3.5 have five. The two figures with five corners, however, also differ. Figure 3.4 is arranged as a pentagon, and Figure 3.5 is arranged as a quincunx (five-pointed star).

Looking carefully at **Figure 3.3**, we can see clearly what is afoot. We can see the changing turns and the end of the line where the turn is through 180 degrees. The starting angle is 0 and so the turns go 0,20,40,60... 140,160, 180, 200, 220 ...

When the turn is through 180 degrees, that is a complete reverse. The turn of 200 is back in the same direction as before (160 + 200 = 360). The turtle retraces its steps, goes past where it started, does a repeat performance — and then shuttles.

Figure 3.3

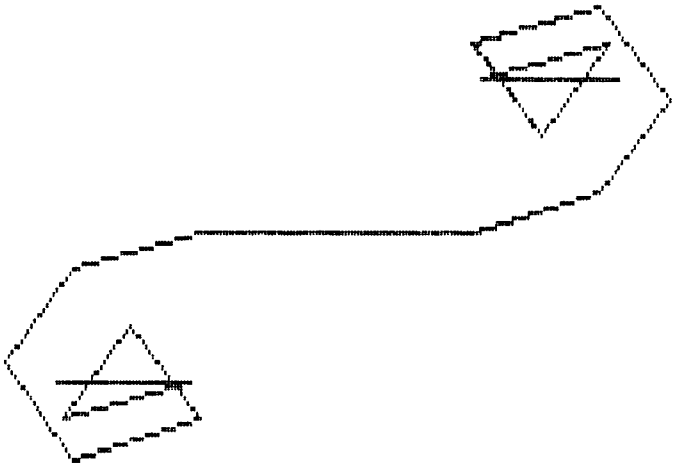


Figure 3.4

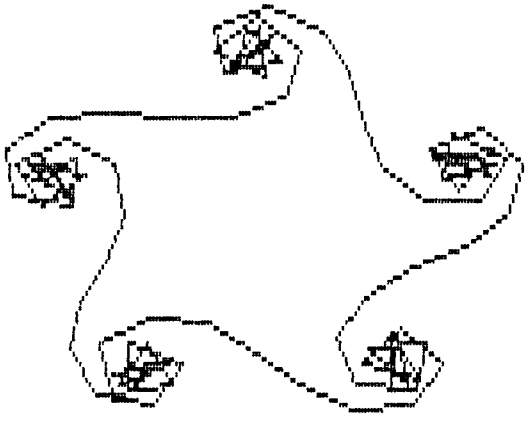


Figure 3.5

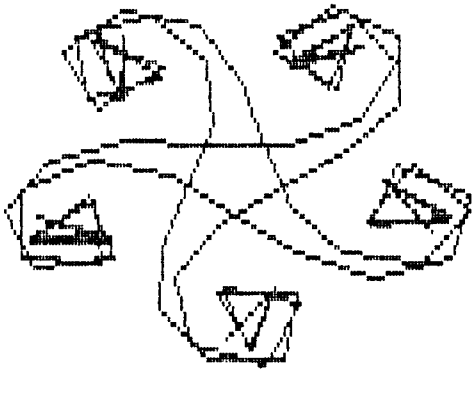
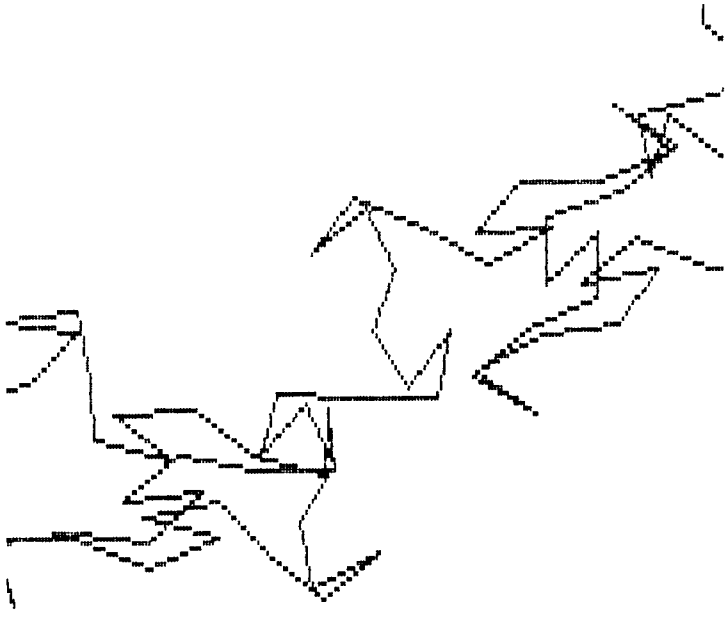


Figure 3.6

Figure 3.7



Figures 3.6 and **3.7** go together. They are the same curve, only in the case of 3.7 we increased the value of DI by quite an amount. The portion under magnification is that in the middle, and what appears as a splodge on Figure 3.6 is revealed as delicate lace work.

This shows the effects of resolution. We cannot get beyond a certain condensation of shapes before they lose their integrity. The angles for these curves are A0 = 1 and A1 = 79, and I then investigated changes in the second parameter, to see what effects there were.

Figure 3.8

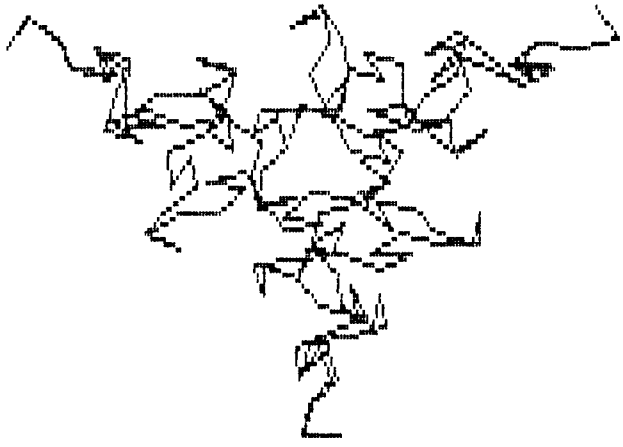


Figure 3.9

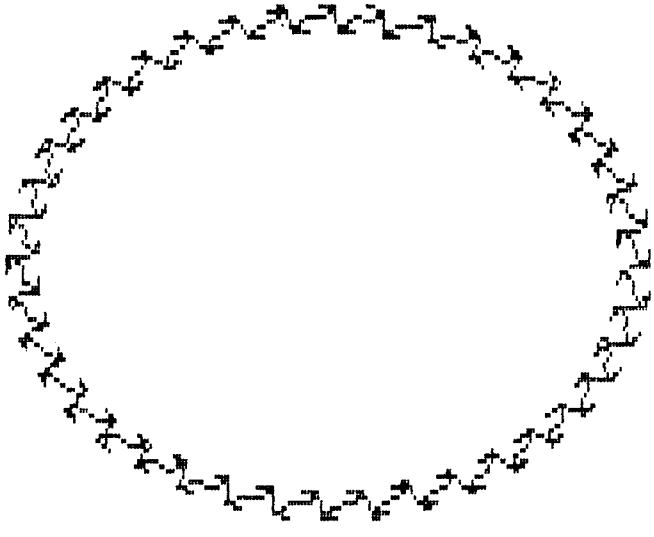
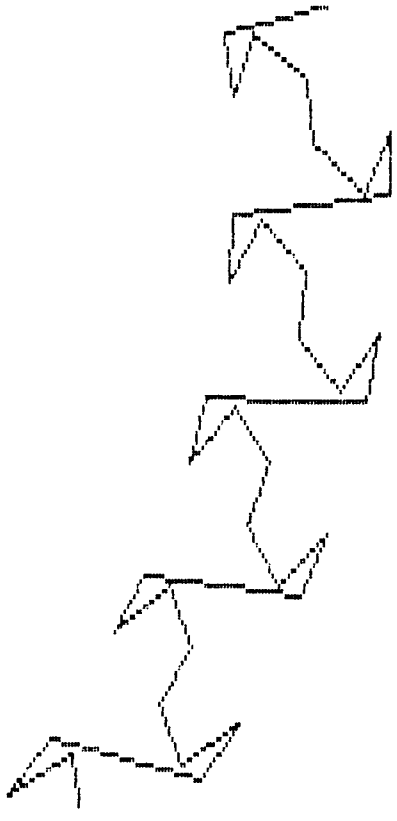


Figure 3.10



My favourite figure (which took ages) is **Figure 3.8** (1 and 78). This has the same start, but a different increment in angle, and it looks rather dissimilar. Figure 3.9 is different again.

When DI was small enough, the values 1 and 80 produced the ring of things, which is **Figure 3.9**. Intrigued to find out what were these things, I magnified the plot, and cut out side wrap round (**Figure 3.10**). I produced the blow up ring of Egyptian hieroglyphics.

I challenge any person to produce these effects on the C64 without using some form or other of turtle graphics. If the turtle graphics routines can cope with curves such as these, what cannot they perform? (Especially if we could speed them up).

CHAPTER 4
Multicolour Turtle Graphics

When we move to a multicoloured turtle graphics system, we find that we are limited to three turtles. We can draw lines in three different colours, and so we identify each colour with a turtle.

Organizing turtles

In the simple high resolution turtle graphics system, the turtle had a position, and it had a direction. It moved from its last position to its new position. The turtle might have its pen up, or — more usually — the pen was down, and lines were drawn.

If we have three turtles, then each turtle has a position and a direction. Each turtle moves from its own last position to its own new position. We have to provide some way of keeping abreast of each turtle's moves and turns.

```
10REM
20REM ------------------------------------
30REM
40REM MULTI-COLOUR TURTLE GRAPHICS :INIT
IALIZE
50REM
60REM ------------------------------------
70REM
20000 DIM XL(2),YL(2),XN(2),YN(2),AN(2),
PE(2) : REM COORDS, ANGLES, AND PENS
20010 FOR I=0 TO 2
20020 XL(I) = 160 : XN(I) = 160 : REM X
COORDS
20030 YL(I) = 100 : YN(I) = 100 : REM Y
COORDS
20040 PE(I) = 1 : REM PENS DOWN
20050 GOSUB 10000 : REM INITIALIZE GRAPH
ICS
20060 RETURN
```

We do this by the use of arrays. XL and YL are arrays which keep track of the last place the turtle reached. The arrays XN and YN are used to find where the turtle has to go (this is not really necessary, but useful for decoration).

Each turtle's angle is stored in array AN, and the condition of each pen is stored in array PE.

All arrays have three elements (for three turtles) and they correspond to turtles 0, 1 and 2. Though not explicitly mentioned in the initialization routine, the turtle number is stored in TN (always TN for the currently operative turtle). By default in C64 BASIC, TN is automatically set equal to 0 when the program is run. In C64 BASIC all variables are assumed to be zero if they have not been used before.

Both the last and new coordinates for the X axis are set to 160, and those for Y are set to 100. Each pen condition is made 1 (otherwise the default would occur, ie 0). The high resolution initialization routine for multicolour graphics is called (see Appendix I), and then the system is ready for use — ready, that is, if the turtle routine to draw a line is present.

```
10REM
20REM ------------------------------------
30REM
40REM MULTI-COLOUR TURTLE GRAPHICS : DRA
W LINE
50REM
60REM ------------------------------------
70REM
21000 XN(TN) = XL(TN)-DI*SIN(PI*AN(TN)/1
80) : REM NEW X COORDS
21010 YN(TN) = YN(TN)-DI*COS(PI*AN(TN)/1
80) : REM NEW Y COORDS
21020 IF PE(TN) = 0 THEN RETURN
21030 NY = YN(TN) : NX = XN(TN) : REM SA
VE COORDS FOR LINE
21040 LY = YL(TN) : LX = XL(TN)
21050 CR = TN+1 : REM COLOUR CODE IS ONE
 MORE THAN THAT OF TURTLE NUMBER
21060 GOSUB 12000 : REM DRAW LINE
21070 XL(TN) = XN(TN) : YL(TN) = YN(TN)
: REM SAVE LAST COORDS
21080 RETURN
```

To draw a line in multicolour high resolution graphics, the drawing routine (at line 12000) needs to be provided with the starting and finishing

coordinates, and with the colour number of the line which is to be drawn.

This colour number (which goes from 1 to 3) is set equal to one more than the turtle number (line 21050) — eg turtle 1 has line colour 2. The function of subroutine 21000 is effectively that of subroutine 12000 for the simple two-colour turtle graphics (see Appendix H).

One item worth noting is that XN and YN are not necessary, but that they have been used in case the system is further extended to incorporate new functions and routines.

As subroutine 12000 uses the labels LX, LY, NX, and NY, we have to convert from the individual turtle coordinates to abstract coordinates for subroutine 21000. That is, we change YN(TN) into NY (and so forth).

Swinging squares

Here is a very simple program. Unfortunately, the effect of colour is lost on a black and white screen dump, so I will not try to produce a representation of the final design.

This program created several problems, because it did not work and it was very difficult to ascertain why. This final version works

```
40 GOSUB 20000 : REM INITIALIZE
50 DI = 50 : REM THE DISTANCE TO BE MOVED
60 FOR TN = 0 TO 2 : REM FOR EACH TURTLE
70 AN(TN) = 120*TN : REM EACH TURTLE HAS A STARTING
ANGLE (IE 0, 120, AND 240)
80 GOSUB 21000 : AN(TN) = AN(TN) + 90 : GOSUB 21000 : AN(TN) =
AN(TN) + 90 : REM HALF A SQUARE
90 GOSUB 21000 : AN(TN) = AN(TN) + 90 : GOSUB 21000 : AN(TN) =
AN(TN) + 90 : REM THE OTHER HALF
100 NEXT TN
110 END
```

The screen goes blank, then becomes yellow, and then a square is drawn (just as it always is). When that square is finished, another square starts — in a different colour. The new square is also at an angle of 120 degrees to the first square.

After this square is drawn, a third square is produced, in a new colour, at a further 120 degrees counterclockwise. So, we have three squares being rotated at 120 degrees at a time, in three different colours.

At line 60, in the program, we use the standard turtle number (TN) as a loop counter. The loop starts at 0 and goes to 2, so that then each turtle will be addressed. Each turtle is given an initial orientation of 120 degrees from each of its fellows (ie TN*120).

The square at lines 80 and 90 is given as four separate moves. This is not normally to be recommended, but when I used a loop instead of four separate moves there was an error. This was very difficult to track down, because having entered multicoloured mode all information about errors was untraceable, unless parts of the program were removed. Removing parts of the program for error tracing was usually of no help, because the program then worked. Finally, I think I solved the problem: there were too many FOR loops for C64 BASIC to cope.

As soon as I stopped the square being a loop, and either had four separate commands or an IF THEN check on a counter, the program worked.

The moral of this story is — watch your FOR loops, subroutines within loops spell trouble (unfortunately).

A slow sun

For this example, RUNning until completion takes about 20 minutes (I say about because I kept forgetting to check on the program's progress, and missed the program finishing).

The effect is rather pretty, and — once on the screen — can be left to add to the visual environment.

```
300 GOSUB 20000 : REM INITIALIZE
310 DI = 50 : REM SET DISTANCE
320 AN(0) = 0 : AN(1) = 3 : AN(2) = 6 : REM SET INITIAL ANGLES
330 FOR TT = 0 TO 119
340 TN = TT - INT(TT/3)*3 : REM TN BECOMES 0, 1, 2, 0, 1, ...
350 GOSUB 21000 : REM DRAW LINE FROM "CENTRE" AT AN
ANGLE AN(TN), LENGTH DI
360 AN(TN) = AN(TN) + 9 : REM LINES STEP ROUND IN
INCREMENTS OF 9 DEGREES
370 XL(TN) = 160 : YL(TN) = 100 : REM MOVE LINE START BACK
TO CENTRE
380 NEXT TT
390 END
```

The program is (of course) simple really: lines of three different colours are drawn, radiating from the centre. We draw a many-coloured sun, full of interesting interference effects. The only problem is the slowness of the drawing.

To the progam. We initialize by calling 21000, setting the distance to a common value of 50, and by setting the starting angles, so that the turtles progress in units of three degrees from each other.

As there are 360 degrees in a full turn, and we have increments of three degrees with our turtles, we make 0 to 119 moves (ie 120 = 360/3). The

Turtle Number is calculated by the module (to base 3) of the number of lines (ie TT).

The line is drawn, the turtle is turned through nine degrees, and returned to the centre. We then progress to the next move, where this program is speeded up and slightly modified.

Faster prettiness

The last program can be speeded up by reducing the number of lines to be drawn, and this is what this new program does. It does slightly more, however.

This program is essentially the same as the previous one, altering the size of steps round and moving a variable distance at each step (it has more than a few resemblances to one of the two-colour turtle graphics programs).

The initial distance is now set to DC. The total number of lines has been reduced by a factor of three, so all turns and increments are as before, with the difference being a factor of three.

In the drawing — as DI is now no longer a constant — we have to give DI a new value on each occasion. DI becomes a function of the ABSolute value of the SIN of the angle. The reason for using the absolute value will become clear if you try to RUN the program without the use of ABS. There are minus moves, so the turtle progresses backwards.

The alterations to the previous program produce this new listing

```
300 GOSUB 20000
310 DC = 50
320 AN(0) = 0 : AN(1) = 9 : AN(2) = 18
330 FOR TT = 0 TO 39
340 TN = TT - INT(TT/3)*3
350 DI = ABS(DC*SIN(PI*AN(TN)/180))
360 GOSUB 21000
370 AN(TN) = AN(TN)+27
380 XL(TN) = 160 : YL(TN) = 100
390 NEXT TT
999 END
```

This simple technique can be extended to produce other, more complex, displays almost as simply as the ordinary two-colour turtle graphics routines.

CHAPTER 5
Further Curves

In the *Programmer's Reference Guide* there is a routine to draw a sine curve in high resolution graphics.

In essence, the program calculates the sine equivalent to every X coordinate, that is, 320 different values of the sine function. Using high resolution graphics can be slow, but this makes it even slower.

In Appendix H, I give a short procedure to calculate a parabola by a much quicker method. This is the method we should use to draw the sine curve. If we do not need even to use turtle graphics we just use the high resolution routines to draw straight lines between fewer coordinates.

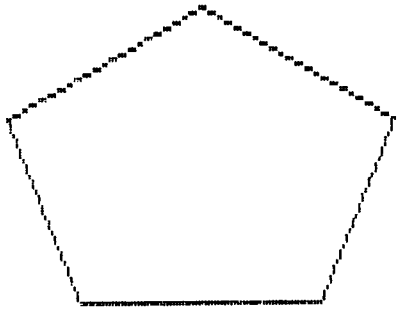
Our eyes are not perfect, and neither is our screen display, approximations are often good enough.

To illustrate this, draw a circle: or, rather, draw a circle in high resolution graphics. Do not use the formula for a circle — use this program:

```
10 INPUT PR,SD
20 GOSUB 20000
30 A1 = 360/SD
40 DI = PR/SD
50 FOR II = 1 TO SD
60 GOSUB 21000
70 AN = AN + A1
80 NEXT II
90 GOTO90
```

Figure 5.1 shows what happens when we enter a value of 5 for the SD parameter. We draw a pentagon — a five-sided figure.

Figure 5.1



We keep the perimeter (PR) constant and alter the number of sides (SD) to 9, 15, 20, and 30 (the last in **Figure 5.5**). When we get to the 30-sided polygon, it is a circle as far as our eyes can tell. To draw a circle this way is also much quicker...

Figure 5.2

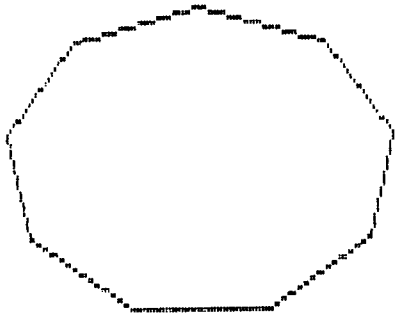


Figure 5.3

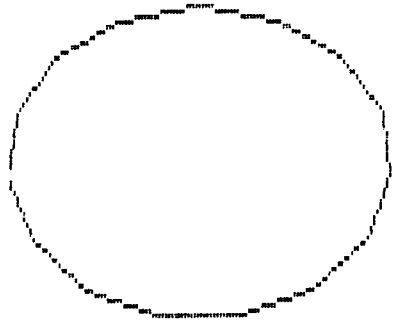


Figure 5.4

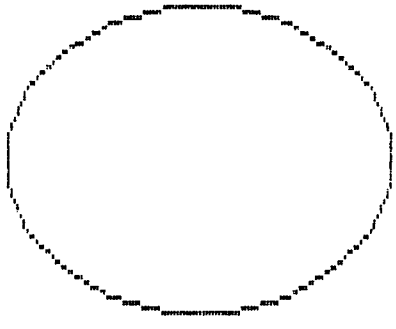
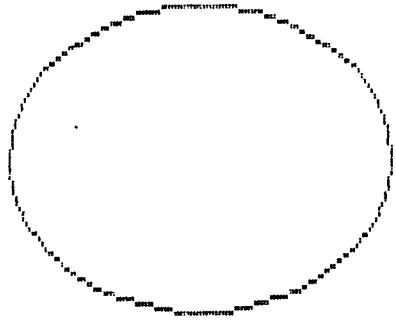


Figure 5.5



What you make of turtle graphics (and not just the plain high resolution version) is up to you. But then that is true of everything.

SECTION 2

Further Steps

APPENDIX A
PEEK and POKE

The C64 is a marvellously flexible and extendible machine, though it may not immediately be apparent to the new user.

This flexibility and extensibility is because the C64 hardware is very powerful, and because it is a very transparent machine. By transparent I mean that it is possible to access and change most of the C64's workings, because Commodore provide the information in some form or other.

One disadvantage of the creditable transparency of the C64 is that some features — which are standard on other computers — are not available. One key lack is, of course, that of a PLOT command. This book is designed to remedy that deficiency, and extend that one command somewhat further.

Though the C64 is potentially so powerful, with an excellent processor plus associated graphics and sound chips, when we go beyond simple programming, things suddenly become much more difficult. To use the C64 to something more like its full potential (and what a potential), then the user has to become more knowledgeable about the machine's workings.

The appendices to this book are designed to supplement the text by showing in fair detail some of the reasons behind some of the more important (and more difficult) characteristics of the C64. Some of the explanations are not just peculiar to the C64 (though all the locations are specific), some are of general topics — true for many machines, especially those by Commodore. It is possible to use the routines given in the main text without references to the appendices, but the reasons behind the routines will not be as clear.

I hope that these appendices will be useful generally as well as in the mere programming of graphics. The appendices are principally orientated towards general programming techniques and hardware details, with the special field of application being that of graphics.

One of the most noticeable characteristics of the C64 in more advanced programming, for those not used to the Commodore approach, is the extensive use of the PEEK and POKE commands for driving the machine. The second most noticeable characteristic is the use of logical operators to alter the values stored in the locations named by the extraordinarily frequent PEEKs and POKEs.

This appendix discusses the use of PEEK and POKE commands; the next appendix explains about hexadecimal and binary notations, and complement arithmetic; and the third appendix shows both how logical operators work, and how they can be used to alter values.

POKEing away

The *Commodore 64 Microcomputer User Manual* (*MUM*) has a short section on PEEKs and POKEs (pages 60–62) and two little additional sections on pages 123 and 126 and, though *MUM* is not supposed to be a definitive programming manual — they want you to buy further books from Commodore — this amount of detail is not sufficient.

The form of the POKE statement (and note, on page 123, POKE is listed as a statement) is

```
POKE address, bytevalue
```

The address is a value between 0 and 65535 (or $0 to $FFFF in hexadecimal notation, and 0 to 1111111111111111 in binary notation). The address refers to a location in memory. *MUM* does not give a proper chart showing the content and arrangement of memory, but *PREG* (the Commodore 64 *Programmer's Reference Guide*, the extra manual you are encouraged to buy) has many such charts. The charts are usually known by the name memory maps, because they map out what lies where in the computer's memory — they map out the memory.

To find your way around memory requires fairly complete memory maps, if you are to know what is where in the memory, otherwise we are POKEing around in the dark. *MUM* does not have proper, full, memory maps, though there are maps for screen memory and colour memory. The point I am trying to make is that you should be careful how you use POKE, unless you have a memory map at hand. If you only have *MUM*, then only use POKEs you have been recommended to use.

If you are a *MUM*mer only, then you are safe to use the patch of memory from 40960 ($A000 in hex) to 49151 ($BFFF in hex). The reason you are safe is that this is the patch of memory given up to the CBM BASIC ROM. ROM means Read Only Memory and, as you can only read from those locations, to POKE in a value does not affect anything. (*PREG*gers can compare pages 212 and 262.)

There are other safe places — and some safe places where you can actually change values — so turn to *MUM* page 63. The diagram there shows the layout of the screen on the C64. There are 40 characters across the screen (0 to 39 inclusive), and 25 lines down (0 to 24 inclusive), giving a total of $40 \times 25 = 1000$ characters on the screen at the same time. Note at this stage the very common way of numbering the first element, as number 0.

Each position on the screen is not activated by some mysterious magic, each has to have tabs kept on it by the computer. If the computer forgot what was at a position, nothing would appear — you cannot get something for nothing.

The picture on a television screen is not static, for it is continually being built up from the top of the screen. There is a stream of electrons banging away at the screen, moving across then down a trifle, until the bottom of the screen is reached (the screen has been scanned). When the bottom has been reached, the stream of electrons starts at the top again, and moves down — a forgetful computer would not know what to tell the stream.

Each of the locations shown, from 1024 ($0400) to 2023 ($0737) is a token, which stores information about what is supposed to be showing on the corresponding location on the screen. The screen is said to be memory-mapped. This mapping is little to do with cartography, but refers to the mathematical use of the term mapping. In this case the mapping means a matching, and the match is between each location on the screen and the corresponding (and unique) location in memory.

Turning over the page in *MUM* (to page 64), shows another set of memory locations (55296, $D800, to 56295, $DBE7), the locations remind the computer what colours are on the screen at the corresponding positions. Each location on the screen has a corresponding location in memory to denote the shape at that point, and another unique memory location to specify the colour of the shape.

If the colour on the screen at a certain position is stored in the computer, the colour must be stored as a number. The numbers are given on page 61 of *MUM*, and so first we will change the character whose colour is stored in the location corresponding to the top left of the screen.

The memory location is 55296 ($D800), and if we change the colour to black this corresponds to a value of 0

```
POKE 55296,0
```

which changes the colour of the letter in the top left corner to black.

Hold down SHIFT and CLR/HOME. On page 133 of *MUM* you will find (in the column headed POKE) the number 65. Under the heading SET 1 there is a spade, and under SET 2 there is the letter A. We will POKE this spade into the bottom left corner, coloured black:

```
POKE 56256,0 : REM READY FOR BLACK
POKE 1984,65 : REM THE SPADE SHAPE
```

and if we now

```
POKE 56256,4
```

there is a purple spade.

Holding down the Commodore key (C=) and SHIFT changes the spade to an upper case A, and the other letters to lower case — this is the change from SET 1 to SET 2. The change can be reversed by SHIFT and C=. Return to the usual (default) SET.

Next try

```
POKE 53280,0
POKE 53281,0
```

and we produce a sombre, all black, picture. Location 53280 ($D020) gives the border colour, and location 53281 ($D021) gives the background colour (*MUM* page 60). As the background is black, to

```
POKE 56256,0
```

makes the spade disappear — it is now the same colour as the background. If you carefully move the flashing cursor down by using RETURN, when the bottom line is reached the spade appears in the cursor, and disappears, and appears....

The bytevalue

Using the screen to demonstrate the use of POKE is truly a graphic demonstration — but there are two parts to a POKE, and we have examined only the first part (the first parameter), the address. The second parameter is the one I called the bytevalue. What does that mean?

We have already used values for the second parameter (otherwise POKE would not work), but what are the characteristics of a bytevalue? Rather than POKE in the screen or colour memories we will use another safe portion of memory. This is a patch of memory which extends from location 251 ($00FB) to location 254 ($00FE), and those who have *PREG* can turn to page 316, where it is described as 'Free 0-Page Space for User Programs'.

A careful examination of the C64 memory map (pages 310–320 of *PREG*) shows other unused patches of memory, but this portion is sufficient for our purposes. First switch off and then switch on again. Second, use what is described as a function by *MUM* (page 126) — the PEEK —

```
PRINT PEEK(251)
POKE 251, 255
PRINT PEEK(251)
```

to which the response is normally 0 to the first PRINT, and always 255 to the second PRINT: if the second PRINT is not 255, then something is wrong with your entry (most likely) or the C64 (most unlikely).

Trying to

```
POKE 251, – 1
POKE 251,256
```

will give ?ILLEGAL QUANTITY ERROR for both. The bytevalue has to be in the range of 0 to 255 ($0 to $FF), and these are the limits on the value of a byte.

A byte is the content of a location (the address of which can extend from 0 to 65535, $0 to $FFFF), and a byte is made up of the equivalent of 8 bits. A bit is a binary digit, a 0 or 1, and on pages 76–78 of *MUM* there is a short section on binary numbers. Binary numbers are important in many ways, but their applicability is most readily seen in the production of user-designed shapes, and in designing sprites, and — we will find — in the POKEing of values into locations.

Appendix B examines binary arithmetic in more detail, but for our purposes we need to know that an 8 bit binary number can take values from 0 to 255 (or $0 to $FF, or 0 to 11111111 in binary). This means that we can only store numbers between 0 and 255 by use of POKE (on some computers, any number can be stored, but the value left in the byte will still only be between 0 and 255). The number 255 is very important for many reasons: see, for example, Appendices E and F of *MUM*.

There is an interesting relationship between the range of addresses and the range of bytevalues. From 0 to 65535 is a total of 65536 different values (including 0), and from 0 to 255 is 256 different values. 256*256 = 65536, 256 is $100 and 100000000 in binary, and 65536 is $10000 and 1000000000000000 in binary. The address of a location is stored in two bytes, which explains the value 66535, which also is a number which occurs in many contexts. 1K is equal to 1024, and the 64K of the C64 is equal to 65536.

Owners of *PREG* might like to verify that assertion by checking the memory map on pages 311–312 for memory locations 43 ($002B) to 56 ($0038). The locations contain pairs of bytes, and they are all pointers to locations in memory. There are other pointers scattered through the rest of the memory map, but these pointers show up clearly because they are grouped together.

Switch off, and switch on again, try

```
PRINT PEEK(1224)
```

to which you get the response 18. Examining the screen display codes (*MUM* page 133), we find that 18 corresponds to R in SET 1. Comparing the location given in the PEEK with the screen memory map on page 63 of *MUM* shows that location 1224 corresponds to the leftmost position on the sixth line of the screen. On power-up that line is occupied by READY, the first character of which is R.

For further elucidation try to enter

```
FOR I = 0 TO 4 : PRINT PEEK(1224 + I) : NEXT I
```

and at which you find the numbers 18, 5, 1, 4, 25, are printed out — corresponding to the characters R E A D Y (*MUM* Appendix 3). To make READY five different colours (see *MUM* page 64)

```
FOR I = 0 TO 4 : POKE 55496 + I, I : NEXT I
```

(in immediate mode) and we get a five-coloured READY.

As we have switched on, try to find out which number represents the background colour. This means we have to PEEK at location 53281 to find out (*MUM* page 60)

```
PRINT PEEK(53281)
```

and we then discover that the result is 246. 246 is not a number from 0 to 15 (*MUM* page 61) — what has happened? If we enter

```
PRINT PEEK(53281) AND 15
```

we produce the rather more acceptable answer 6, ie the darker blue. Very strange. Furthermore

```
PRINT PEEK(53280), PEEK(63280) AND 15
```

produces 254 for the first and 14 for the second. The number 14 is acceptable because it is used to represent the lighter blue.

MUM (page 62) says that 'By entering AND 15 you eliminate all other values except 1–15, because of the way color codes are stored in the computer. Normally you would expect to find the same value that was last POKEd in the location.' Appendix B discusses the use of binary (and hexadecimal) numbers to help clarify that statement, and Appendix C gives routines to make such manipulations of bytevalues rather more simple.

At the moment it is sufficient to remember the limits on the values that can be stored in a byte (ie 0 to 255), and how the size of memory is determined by what can be stored in two bytes — ie numbers from 0 to 65535.

APPENDIX B
Investigating Arithmetics

Hexadecimal arithmetic does not appear in the Index to *MUM*, but binary arithmetic does appear (the discussion is on pages 75–78).

Knowledge of hexadecimal arithmetic is not essential to BASIC programming of the C64 at any level, partly because there is no way in C64 BASIC to input hexadecimal numbers — unless one writes special routines to so do.

The use of hexadecimal notation becomes far more important when we begin to use machine code. As I am not going to use any machine code routines, my discussion of hexadecimal will be rather short only, and will follow after the discussion of binary arithmetic — which is, however, important in BASIC.

Decimal numbers

The ordinary (decimal) number 4567 is capable of being split up in this way

4567 = 4000 + 500 + 60 + 7

which is no more than the isolation of the thousands, hundreds, tens, and units.

One thousand is equal to 1*10*10*10, or (in a shorthand notation) 1E3 — where the number after the E gives the number of tens multiplied together. Another way of writing the number is thus

4567 = 4E3 + 5E2 + 6E1 + 7E0

Where the final number 7E0 indicates that the value is 7 times no tens (*MUM* does not really discuss the use of E, though it is introduced on pages 26–27, and *PREG* has a small amount on pages 6 and 11). Actually, if you try

```
PRINT 8E-1
```

you find the answer is 0.8.

The minus in front of the number of tens indicates that, instead of multiplying by that number of tens, we divide by it. (The reverse of plus is minus, and thus the reverse of multiply is divide.)

The number

```
PRINT 4.5678E3
```

is 4567.8. and so .5E3 must be the same as 5E2, with .007E3 being the same as 7E0 (ie 7). E is a convenient way of showing place value for decimal numbers. Why do we call ordinary numbers decimal numbers?

As we move from left to right down each of the separate digits, the place value of each digit decreases by 10. In other terms, the value after the E decreases by one as we move from left to right. The key number is 10 (thus decimal), and this is why the individual numbers are called digits — a digit is a finger or thumb, and we have ten fingers or thumbs (generally speaking).

Another way of writing 4567.8 is

```
PRINT 4*10^3+5*10^2+6*10^1+7*10^0+8*10^-1
```

and the up arrow ^ is called the exponentiation operator (*MUM*, page 25; *PREG*, page 12). The value to the left of the up arrow is called the base, and it is multiplied by itself the number of times indicated by the value on the left (the exponent). As the base used for 4567.8 is 10, a decimal number is also called a number to base 10.

A more exact definition is that the base is raised to the power of the exponent. Try the expression

```
PRINT 9^(1/2)
```

that is, print the value of 9 raised to the power of a half. The answer is 3. The power of a half is equivalent to taking the square root of the value: the square root of 9 is 3 (because 3*3 = 9).

The same result could be found by entering

```
PRINT SQR(9)
```

which also gives the answer 3, or, rather more mysteriously,

```
PRINT EXP(LOG(9)/2)
```

another answer of 3. It could be that this last result is merely fortuitous, the result of mere chance: it is not, however, mere chance.

The function EXP means raise the constant 'e' to the power of the following value: eg EXP(2) means e^2. The function LOG means take the natural logarithm of the following value, and LOG(EXP(X)) = X (see *MUM* page 126, and *PREG* pages 46 and 61).

The calculation of X^Y for values of Y which are not integers (ie whole numbers) is via the use of EXP(LOG(X)*Y). This explains why it is possible

to calculate (−3)*(−3) as (−3)^2, but to try to calculate (−3)^2.1 produces an ?ILLEGAL QUANTITY error. When the exponent is not an integer, the BASIC translator has to take the LOGarithm of a negative number — and that is impossible.

Are the above answers exact? Try

```
PRINT 9^(1/2)-3, SQR(9)-3
```

to which both answers are 9.31322575E−10. This implies that the answers to both calculations are not exactly 3: there is some slight arithmetical/computational error in the calculation of both.

To any person who has used a calculator this slight inaccuracy will come as no surprise. On almost any calculator the square root of 1111, when multiplied by itself, does not equal 1111. On my calculator the square root of 1111 is 33.331666, and when 33.331666 is multiplied by itself (squared) the resut is 1110.9999. To explain how the errors occur with the C64, we need to look at binary arithmetic.

Binary numbers

Binary numbers are numbers for which the base is 2. The binary number 11001 can be split into

```
11001 = 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0
```

(compare the composition of 4567). As 2^4 = 2*2*2*2 = 16, then

```
11001 (binary) = 16 + 8 + 0 + 0 + 1 (decimal)
               = 25 (decimal)
```

and so we can see that the decimal number, corresponding to the binary number 11001, is 25. This can be checked on the C64 by

```
PRINT 2^4+2^3+2^0
```

to which the answer is 25.

If the base is 2 this means that there are only two different numerals (ie 0 and 1), just as with base 10 there are ten numerals (0 through 9). These two values can correspond to a high as against a low voltage, or an off/on switch or relay: computers use base 2 arithmetic, because they use high or low voltages to record values.

On the C64 the basic building block used to store information/values is the byte, mentioned in the previous appendix: a byte consists of eight bits. A bit is short for a binary digit, that is, a 0 or 1. If the byte has all bits set to zero, ie 00000000, then its value must be zero. If all bits are set to unity, ie

11111111, then the byte must have the value 255.

The result of 255 (as I noted, a very common number in computing) is derived by noting

```
2^7 = 128
2^6 = 64
2^5 = 32
2^4 = 16
2^3 = 8
2^2 = 4
2^1 = 2
2^0 = 1
```

and the total is 255. (It is worth comparing the section in *MUM* — pages 70–73 — on the definition of sprites, in which there is a similar explanation.)

At the end of Appendix A, I quoted from *MUM* (page 62), concerning AND 15. We are now in a position to see what is so special about 15: 15 = 8 + 4 + 2 + 1, and so as an eight bit binary number (a byte) it is 00001111. The number 246 (see Appendix A) is 128 + 64 + 32 + 16 + 0 + 4 + 2 + 0 or, as a byte, it is 11110110. To calculate 246 AND 15, we convert both numbers into their binary equivalents:

```
246 = 11110110
15  = 00001111
```

and then take each position in turn, multiplying the bits together. This produces

```
246 AND 15 = 00000110 = 6
```

(logical operators will be considered in detail in the next appendix). It is easy to see that to AND with 15 isolates the rightmost four bits (also called the least significant bits).

If the C64 calculates by using binary arithmetic, perhaps the approximation we found (ie 9.31322575E–10) is related to powers of 2? **Figure B.1** shows powers of 2 for negative values. Just as 10^(–1) = 1/10 = .1, we find that 2^(–30) = 9.31322575E–10 (see Figure B.1). The error of approximation we found is thus exactly related to powers of 2, and therefore may be due to operations of binary arithmetic.

Try this:

```
PRINT 4.65661287E-10+1-1
```

to which the answer is 4.65661287E–10. Now try

```
PRINT 2.32830644E-10+1-1
```

to which the answer is 0. How is this to be explained?

Figure B.1

NEGATIVE POWERS OF 2

Power	Value
0	1.00000000E0
-1	5.00000000E-1
-2	2.50000000E-1
-3	1.25000000E-1
-4	6.25000000E-2
-5	3.12500000E-2
-6	1.56250000E-2
-7	7.81250000E-3
-8	3.90625000E-3
-9	1.95312500E-3
-10	9.76562500E-4
-11	4.88281250E-4
-12	2.44140625E-4
-13	1.22070312E-4
-14	6.10351562E-5
-15	3.05175781E-5
-16	1.52587891E-5
-17	7.62939453E-6
-18	3.81469727E-6
-19	1.90734863E-6
-20	9.53674316E-7
-21	4.76837158E-7
-22	2.38418579E-7
-23	1.19209290E-7
-24	5.96046448E-8
-25	2.98023224E-8
-26	1.49011612E-8
-27	7.45058060E-9
-28	3.72529030E-9
-29	1.86264515E-9
-30	9.31322575E-10
-31	4.65661287E-10
-32	2.32830644E-10

When the C64 calculates the answer to each of these lines, it starts at the left and builds up a cumulative sum as it goes along. When 1 is added to

4.65661287E−10, the resulting number is still seen by the C64 as different from 1, so that, when 1 is subtracted, the answer printed out is 4.65661287E−10.

When 1 is added to 2.32830644E−10, the C64 thinks that the result is 1 — compared to 1, 2.32830644E−10 is too small to be distinguished by the computer. When 1 is then subtracted the result is the answer 0 (because 1 − 1 = 0). No computer is ever totally accurate in all circumstances — and the C64 is no exception.

The smallest value that the C64 can distinguish from 1 is 4.65661287E−10, and that is a difference of 2^0 and 2^(−31), which is equivalent to the left and right bits in a 32 bit number. The C64 stores fractional numbers (ie those usually called floating-point numbers) in four bytes — 32 bits. The relative size of the number (a trifle like the power) is stored in one additional byte — 8 bits. A floating point number thus takes up five bytes in memory.

Base conversions

In the previous section, when we converted 246 into a binary number, we performed a calculation which is eminently computable.

Begin to think how we could compute a binary number, and, furthermore, suppose that we are trying to emulate (ie copy) an integer in C64 BASIC. An integer in C64 BASIC is two bytes long, and an integer variable is indicated by the % postfix. X is a floating-point variable, whereas X% is an integer variable (and X$ is a string variable).

MUM is fairly quiet on the topic of integer variables (eg pages 112−113), and *PREG* (pages 5, 7−9) is only slightly more helpful. In *PREG* we are told that integers are stored in memory as two-byte binary numbers, and that integers in C64 BASIC are whole numbers between −32768 and +32767.

If you think back to Appendix A, we discovered there that a two-byte binary number could take 65536 different values (usually from 0 to 65535). In this case an integer still takes 65536 different values, but the values go from −32768 to +32767. We will have to wait to find how this new arrangement of values works.

Return to that value 246. We will start at the right of the binary number, the least significant bit. If the right bit is a zero then the number is even, and if the right bit is unity then the number is odd. 246 is even, so that the rightmost bit is zero. We have, therefore,

xxxxxxxxxxxxxxx0

fifteen unknown bits, and one known (a 0).

If we now consider the fifteen remaining bits (shown by x), they correspond to the number 246/2 = 123. The number 123 is odd, so the rightmost bit is unity, thus,

xxxxxxxxxxxxxx10

leaving fourteen bits. These fourteen correspond to the number 61 (ie 123/2 = 61.5, and ignore the .5), and 61 is odd:

xxxxxxxxxxxxx110

The next number to consider is 30, then 15, then 7...

xxxxxxxxxx110110

and ultimately we produce

0000000011110110

in agreement with our earlier result.

The main difference between the two results is that in our earlier version we did not have the eight leftmost zeros. As, however, in the second version we are trying to emulate an integer (of two bytes) we show the full sixteen bits.

We decide whether a number is odd by use of a function,

```
10 DEF FNOD(X) = X - 2*INT(X/2)
```

which works as follows. If X is odd, then X/2 has the fractional part .5 (eg 7/2 = 3.5), the C64 BASIC function INT takes a number and strips off the fractional part, to give the whole number beneath (eg INT(3.5) = 3). If X is odd (eg 7), then the result of the function FNOD(X) is the value 1 (eg 7 - 2*INT(3.5) = 7 - 2*3 = 1). If X is even, then the result of the function is 0.

To try this out, enter the function (say line 10), type RUN to set up the system, and then, in instant mode,

```
FOR I = 0 TO 10: PRINT I,FNOD(I): NEXT I
```

will produce the output

```
0     0
1     1
2     0
3     1
4     0
5     1
6     0
7     1
8     0
9     1
10    0
```

To try next

```
FOR I = – 10 TO 0: PRINT I,FNOD(I): NEXT I
```

will produce

```
 – 10     0
 – 9      1
 – 8      0
 – 7      1
 – 6      0
 – 5      1
 – 4      0
 – 3      1
 – 2      0
 – 1      1
  0       0
```

which is just what we want. If we do not RUN first, there is an ?UNDEF'D FUNCTION error — the system has not yet been made aware of the presence of FNOD.

```
 990 DEF FNHF(Y) = INT(Y/2)
 995 DEF FNOD(X) = X – 2%FNHF(X) : REM CUNNING?
 998 END
 999 REM----------------------------
1000 REM DENARY TO BINARY CONVERSION
1001 REM--------------------------
1100 BI$ = "" : INPUT"DECIMAL NUMBER ";N
1110 FOR I = 0 TO 15
1120 BI$ = STR$(FNOD(N)) + BI$
1130 N = FNHF(N) : NEXT I
1140 PRINT : PRINT "BINARY NUMBER"
1150 PRINT BI$ : RETURN
```

It seems clear that to calculate the binary equivalent of a denary number we have only to repeat the odding on successively halved values of the number (a denary number is another way of describing a decimal number). We might as well have a function to take a half (ignoring fractional parts):

```
20 DEF FNHF(X) = INT(X/2)
```

the reason for using the function being that this is the kind of action we might want to take in other circumstances.

Experimenting with positive and negative values can produce some interesting results, the most interesting of which is that given by

```
PRINT FNHF(-1)
```

to which the response is -1. Seems to be worth some explanation, so try

```
PRINT INT(-3.2)
```

to find that the result is -4: the INT function always produces the whole number less than the supplied value (unless the supplied value is an integer itself).

The final point concerns where we will store the binary number: try this

```
BI$=" EGGS": PRINT BI$: BI$=STR$(2*2)+BI$: PRINT BI$
```

to which the response is

```
 EGGS
4 EGGS
```

and a further line

```
BI$=STR$(2*3)+BI$: PRINT BI$
```

produces

```
6 4 EGGS
```

— but not 64 EGGS.

The string BI$ (a list of characters, *MUM* page 113, and *PREG* pages 6–9, 16–17) is set equal to the characters EGGS (ie a space and the four characters). The content of the string is PRINTed. On to the front of the string BI$ is added the result of 2*2, expressed as characters. 2*2 is equal to 4, but what is added to the front of BI$ are the two characters 4 (ie space and number). The STR$ function gives a string which is identical to the PRINTed version of the number (*MUM* page 128; *PREG* page 87).

PRINTing the new version of BI$ (linked together, or concatenated, from the original version of BI$ and the result of the STR$ function) produces the information that there are 4 EGGS.

Concatenating the BI$, again, with STR$(2*3) gives 6 4 EGGS. Note that the 6 and the 4 are not adjacent: they are separated by a space, almost as they would be if they were displayed by

```
PRINT 6;4;" EGGS"
```

but in this case an extra space intrudes to separate them.

The conversion routine

This routine takes an input decimal number N, and converts the value of N to a binary representation stored in the string BI$. The successive bits in BI$ are built up from the right by repeating odding and halving, so that when the string is PRINTed, the bits are in the correct order.

The two functions previously examined are both used, and could be assumed, but are given prior to the routine for completeness:

```
LIST
   990 DEF FNHF(Y) = INT(Y/2)
   995 DEF FNOD(X) = X - 2 * FNHF(X) : RE
M CUNNING, EH?
   998 END
   999 REM ------------------------------
  1000 REM DENARY TO BINARY CONVERSION
  1001 REM ------------------------------
  1002 REM
  1010 BI$ = "" : INPUT "DECIMAL NUMBER "
: N
  1020 FOR I = 0 TO 15
  1030 BI$ = STR$(FNOD(N)) + BI$
  1040 N = FNHF(N) : NEXT I
  1050 PRINT : PRINT "BINARY NUMBER"
  1060 PRINT BI$ : RETURN
>
```

using the number 246 gives the result we found before (note that the bits are not next to each other, they are separated by one space).

To use the routine, first RUN and second use the subroutine in instant mode, eg:

```
GOSUB 1000
```

and then obey instructions. A good number to try is 1, for which the result is

```
0000000000000001
```

or −1, for which we find

```
1111111111111111
```

surprising as it seems.

Remembering that integers in two bytes extend up to 32767 in the positive direction, we try out 32767, and find

```
0111111111111111
```

One more than that is 32768, or

```
1000000000000000
```

so what is −32768 (the most negative possible value)? It is

```
1000000000000000
```

or exactly the same as 32768. The maximum value possible for 16 bits (two bytes) is 65535, so what does that give? It gives

1111111111111111

or exactly the same as the value for −1.

If you now try out different positive and negative binary numbers you might discern a pattern: another possible idea is to modify the routine so that it does not ask for the input number, and use this

```
FOR N= -5 TO 5: GOSUB 1000 : GET A$ :NEXT N
```

or similar. The pattern is subtle but simple.

If the binary number has a zero in the leftmost bit position, then that number is positive: if the leftmost bit is unity, then that number is minus. The reason why 1000000000000000 is interpreted as −32768 by BASIC (when stored as an integer) is that it starts with a 1, not a 0. If we try to find the binary equivalent of 65536, we discover

0000000000000000

and so 65536 gives the same answer as zero (because 65536 is 1 followed by sixteen 0s, and only the sixteen 0s are stored).

Trying with even larger numbers only succeeds in producing 16 bit numbers which go from 0000000000000000 to 1111111111111111, the numbers circle round like a clock. If 0000000000000000 is zero, and 1111111111111111 plus 1 is zero, then 1111111111111111 must be the equivalent of minus one (−1+1=0).

This system is called two's complement arithmetic, and — as we see — it is a logical way of using negative numbers when we only have positive numbers available. There is a very brief mention of two's complement in *PREG* (pages 63–64).

Hexadecimal arithmetic

A convenient way of displaying sixteen bits at once is to split them into groups of four bits at a time. The number −2 (or 65534) can be displayed as

1111 1111 1111 1110

or possibly — as 1111 is equal to 15 —

15 15 15 14

where each grouping of four bits is treated as a separate entity.

When you think about it, each of those 15s has a different place value — they increase in relative value by a factor (or power) of sixteen. Just as we have base 2 and base 10 arithmetics, we could have base 16, and base 16 would be very convenient for analysing bytes. Two base 16 digits are all that

would be needed, one digit for the first four bits, and one for the other four bits (four bits together are sometimes called a nybble).

It seems rather silly, however, to use clumsy two digit numbers like 15 to describe one base 16 digit, so we invent some new digits. Base 16 arithmetic is usually called 'hexadecimal' arithmetic, and in Commodore manuals a hexadecimal number is normally preceded by a $. The number 65534 (or −2) is shown as $FFFE in hexadecimal notation: the ordering is 0 1 2 3 4 5 6 7 8 9 A B C D E F then 10. 255 is $FF in hex (as the name is often shortened), just as 65535 is $FFFF in hex.

Hexadecimal notation is often utilized in memory maps and similar, because it is simpler and tidier to use; but, in C64 BASIC programming, only decimal notation is used to name locations in memory.

APPENDIX C
Logical Bits

The real amount of useful information concerning graphics in *MUM* is very low, and to perform anything useful *MUM* has to be left behind.

Memory management

The graphics on the C64 are controlled by the VIC-II chip. This chip is a multi-purpose colour video controller device, with 47 registers (ie special locations), which we can access from BASIC by PEEKs and POKEs. The registers have images in the C64's memory, starting at locations 53248 ($D000) through to 53294 ($D02E) — exactly 47 ($2F) locations.

If the memory map in *PREG* (pages 321–323) for locations $D000 to $D02E is compared with the locations in Appendix N of *PREG* (pages 454–455), a one-to-one correspondence can be made.

The main difference is that in the memory map the first 16 locations are given up to sprites, whereas in the description of registers the first 16 are given up to MOBs. MOB stands for Moveable Object Blocks, and is the technical term used for sprites by the manufactures of the VIC-II chip. In Appendix G of *PREG*, which is another list of the VIC-II registers, they are again called sprites.

The VIC chip can control 16K of memory at one time. If there were four VIC chips, then perhaps all of memory could be controlled. As it is, the chip can see only a quarter at a time.

To direct the memory controlled by the VIC chip we make use of what are termed bank select bits. The memory bank to be selected is under the control of another chip, the CIA #2 (Complex Interface Adapter Number 2). To change the part of memory controlled by the VIC chip, we have to perform two operations: first, we have to tell the CIA #2 that we are going to change the memory controlled by the VIC chip; and, second, we have to tell the CIA #2 which part of memory the VIC-II is to control.

The CIA #2 has 16 registers, and they are given in the memory map on pages 331–334 of *PREG* — extending from location 56576 ($DD00) through to location 56591 ($DD0F). The register at location 56578 is what is termed a Data Direction Register — Port A, and has to be activated before we instruct Data Port A to operate (called the 'Peripheral Data Register — Port A'). Data Port A is important because this is the means by which the memory banks are changed (location 56576).

Logical comparisons

In *PREG* (page 101) to change the memory banks, we find these lines

```
POKE 56578,PEEK(56578) OR 3 : REM MAKE SURE BITS 0 AND 1
ARE SET TO OUTPUTS
POKE 56576,(PEEK(56576) AND 252) OR A : REM CHANGE
BANKS
```

where A can take values from 0 to 3. The first line selects the data direction register (port A), and the second line activates the peripheral data register (port A). These lines will be our introduction to the use of logical operators.

The data peripheral register, at location 56576, is a one-byte register, and the eight bits are arranged as in **Figure C.1** (with the leftmost bit being number 7). This description is taken from *PREG* (page 331), and all we need to consider are bits 1 and 0. As far as we are concerned the contents of this location are xxxxxxyy, where x is a bit we do not need to consider thus far, and y is an important bit (ie one we may wish to alter).

Figure C.1 Location 56576

BIT	FUNCTION
7	Serial Bus Data Input
6	Serial Bus Clock Pulse Input
5	Serial Bus Data Output
4	Serial Bus Clock Pulse Input
3	Serial Bus ATN Signal Output
2	RS-232 Data Output (User Port)
1–0	VIC Chip System Memory Bank Select

In the byte at location 56576 all the bits from 2 through to 7 are used for devices other than the VIC-II, so we can ignore all those bits for the moment. Therefore, to prepare the CIA #2 to output the correct bank number via bits 0 and 1 of location 56576, we have to set only bits 0 and 1 of the other register, at location 56578.

We have to set both the bits (0 and 1) at location 56578 to 1, ie we have to set up a pattern of bits which is

xxxxxx11 = 3 (decimal)

and this explains the 3 in the first of the two lines of program. Into location 56578 we POKE the results of using a logical OR on the contents of that location (ie PEEK(56578) with the value 3. If A and B are bits, then the result of the operation A OR B for different combinations of bit values is given in **Figure C.2** (see *PREG* page 68).

Figure C.2 Truth Table for OR

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

Suppose the contents of location 56578 are originally 10101010 (= 170 = PEEK(56578)). Then let us see what are the results of ORing this value with 00000011 (= 3).

```
1 0 1 0 1 0 1 0     =     PEEK(56578)
0 0 0 0 0 0 1 1     =     3
-------------------------------------
1 0 1 0 1 0 1 1     =     PEEK(56578) OR 3
```

If for either of the two bits at any position there is a 1 then the result is a 1 in that position, and the only time that the result is 0 is when both original bits at that position are 0 (this is shown by Figure C.2).

If the result of ORing is POKEd into location 56578, the only difference to the bits are those in positions 0 and 1 — which is just as we wanted. To OR, therefore, is to set specified bits to 1, if a 1 appears in one of the two values being compared. To have a 0 at any position is not to affect the value of the corresponding bit. To sum up: 1 always sets to 1, and 0 does not change.

The first program line, therefore, has set both bits 0 and 1, stored in location 56578, to 1.

The next line is somewhat more complex: there are two logical comparisons. The first comparison is

```
PEEK(56576) AND 252
```

where the binary value corresponding to 252 is

```
11111100
```

that is, the bits 2 to 7 are all 1, and bits 0 and 1 are both 0. The truth table for the AND logical connective is shown in **Figure C.3**, and also compare *PREG* (pages 35–37).

Figure C.3 Truth Table for AND

A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

Let the binary value stored in 56576 be 10101010, and see what happens when we AND with 252, as in the second program line.

```
1 0 1 0 1 0 1 0      =      PEEK(56576)
1 1 1 1 1 1 0 0      =      252
-------------------------------------------
1 0 1 0 1 0 0 0      =      PEEK(56576) AND 252
```

What has happened is that the result of ANDing with 252 is to leave the value of most of the bits in location 56576 as they were; with the only change being the bits at positions 0 and 1 being made equal to 0.

If we call 252 the comparison value, then 252 is composed of comparison bits. To AND with the comparison bit being 1, is to produce a result which is equivalent to leaving the bit in the other value unchanged. To AND with the comparison bit being 0, is always to produce a result of 0. (See Figure C.3).

To set a bit to 1 we OR with 1, whereas to set a bit to 0 we AND with 0. To leave bits unchanged we either OR with 0, or we AND with 1.

The final part of the second line takes the result from the AND, and then inserts bits into positions 0 and 1 by use of OR 3. The mechanism here is the same as that of the first line. For example, let A be 3: the VIC-II will control memory from 0 to 16383 inclusive ($0000-$3FFF) — compare **Figure C.4**.

Figure C.4 VIC-II Memory Control

A (DEC)	A (BIN)	BANK	RANGE (DEC)	RANGE (HEX)
0	00	3*	49152–65535	$C000–$FFFF
1	01	2	32768–49151	$8000–$BFFF
2	10	1*	16384–32767	$4000–$7FFF
3	11	0	0–16383	$0000–$3FFF

Note * The standard character set cannot be used with this portion of memory (memory bank)

A = 3 is 00000011 as an eight bit binary number, and so to take the result of the above comparison (ie PEEK(56576) AND 252) we produce

```
1 0 1 0 1 0 0 0      =      PEEK(56576) AND 252
0 0 0 0 0 0 1 1      =      A        =        3
-------------------------------------------
1 0 1 0 1 0 1 0      =      (PEEK(56576) AND 252)
                            OR A
```

and the bits in positions 0 and 1 are set to the correct values.

Information on locations

There are three other locations which repay consideration when we examine high resolution graphics. These are locations 53265, 53270, and 53272, and details are given in Figures C.5, C.6, and C.7.

Location 53265 is used in this manner:

```
POKE 53265,PEEK(53265) OR 32 : REM BIT MAP ON
POKE 53265,PEEK(53265) AND 223 : REM BIT MAP OFF
```

which, when we realize that 32 is 00100000 in binary and 223 is 11011111 in binary, obviously isolates bit 5. Comparing this to **Figure C.5** reveals that bit 5 of the VIC Control Register at location 53265 sets the bit map mode.

Figure C.5 Location 53265 — VIC Control Register

BIT	FUNCTION
7	Raster Compare (see also 53266)
6	Extended Color Text Mode: 1 to Enable
5	Bit Map Mode: 1 to Enable
4	Blank Screen to Border Color: 0 to Blank
3	Select 24/25 Row Text Display: 1 is 25 Rows
2–0	Smooth Scroll to Y Dot-Position (0–7)

This location does many other things, none of which need concern us at the moment, so just concentrate on bit 4. To set a specific bit to 1, we OR that bit with 1. 32 as a binary number is — as we have noted — 00010000 and, as with OR 3 above, this leaves all bits unchanged except that with a 1. To OR with 32 always sets bit 4 to 1.

The converse is 223 (11101111 as a binary number). If we AND with a bit equal to 1 it leaves the other bit unchanged, but if we AND with a bit equal to 0 the other bit is changed to 0. (Note also that 32 + 223 = 255). To AND with 223 always sets bit 4 to 0.

To enable bit map mode, therefore, we have to OR the contents of location 53265 with 32. To switch off the bit map mode we perform the reverse, we AND the contents of location 53265 with 223.

Location 53270 is another VIC Control Register. The main reason we are interested in this register is because we can use it to set the C64 into a multi-color mode (bit 4, again). The multi-color mode is operative either with ordinary text, or with bit map mode.

The full contents of this location are given in **Figure C.6**, and to enable or disable the multi-color mode we use the same ANDs and ORs as with location 53265.

Figure C.6 Location 53270 — VIC Control Register

BIT	FUNCTION
7–6	Unused
5	ALWAYS SET TO 0!
4	Multi-Color Mode: 1 to enable (Text or Bit Map)
3	Select 38/40 Column Text Display: 1 is 40 Cols
2–0	Smooth Scroll to X Position

Investigating memory control

The final location which need concern us is 53272: the VIC Memory Control Register. This location controls only two things — the relative position of screen memory (in bits 4–7), and the relative location of character memory (in bits 1–3).

When we discussed PEEKs and POKEs, there was a distinction between the screen memory (the actual character on the screen) and color memory (the colour of the character at that position). Screen memory can be moved around in memory (within limits), but color memory cannot be moved (it is fixed at locations 55296 to 56295).

We have seen how it is possible to change the memory controlled by the VIC-II, and what bits 4–7 of location 53272 set is the relative position of screen memory — relative to the start of the patch of memory controlled by VIC-II. The four bits in themselves only go from 0000 to 1111 (ie 0 to 15), which is sixteen different values: VIC-II controls 16K at a time.

If the four bits are 1010 (ie 10 in decimal) this means that the start of the screen memory is 10K forward from the start of the VIC controlled memory. Normally the value of these bits is 0001 which is 1K (ie 1024 bytes), and the default memory controlled by the VIC-II is that which starts at $0000. Normally, therefore, the screen memory should start at 1024 (ie 0 + 1024), and it does.

Bits 1–3 of the byte at the same location control what is termed the character memory. Character memory is where the shapes of the characters which appear on the screen are stored. Each character is composed of eight bytes of information. See Appendix E for further details, but an examination of page 70 of *MUM* might help: the sprite is a trifle like a super character, almost three times as high, and three times as wide.

The bit map occupies 8000 bytes, and is effectively eight times as large as the normal 1000 characters, because of the added rider that each individual bit in each character is separately controllable. To design a character, as against merely displaying the shape on the screen, takes up eight times as much memory (see Appendix E).

Bits 1–3 can take values from 000 to 111 (ie 0 to 7), which is eight

different values. Each increase of 1 in value for these bits thus corresponds to 2K bytes of memory controlled by the VIC-II.

The first 8K bytes controlled by the VIC-II contain screen memory (unless changed), and also contain copies of the C64 character set (the copies are from the character set held in ROM). Effectively this means that character memory (and the bit map) have to be contained in the second 8K of the VIC controlled memory.

To point to this second half of memory, the bit pattern needs to be 100 (= 4), and if the default memory allocation for the VIC-II starts at $0000, then the bit map starts at $2000 (ie 8K = 8196). To set the map up at this position, therefore, we use

```
BM = 8196 : POKE 53272,PEEK(53272) OR 8
```

(compare to *PREG* page 123) remembering that 8 is 00001000 in binary. To OR the contents of location 53272 with 8 is thus to change bit 3 to a 1 — hopefully this may actually be interpreted as a 4 (ie xxxx100x) by the system. What happens, however, if for some reason bits 1 and 2 are not 0?

What happens is that the bit map is put too high in memory for the VIC-II to see all of it, but we still think that the bit map is lower down. We have to mask all those three bits to 0 first, before we actually OR with the number eight. First we AND with 11110001, and then OR with 00001000.

```
BM = 8196 : POKE 53272,(PEEK(53272) AND 241) OR 8
```

(work out why and how this operates).

Figure C.7 Location 53272 — VIC Memory Control Register

BIT	FUNCTION
7–4	Video Matrix Base Address (inside VIC)
3–1	Character Dot-Data Base Address (inside VIC)

The allocation and arrangement of memory is treated in more detail in Appendix D, and Appendix F has a list of binary numbers (with hexadecimal and decimal equivalents), to help in the choice of numbers for logical comparisons.

A final logical operator

There is one final logical operator which it is worth constructing. This is the exclusive OR, which is frequently shown by XOR (sometimes EOR).

The truth table for this operator (a truth table which appears, with others, on page 14 of *PREG* is shown in **Figure C.8**. If A is the comparison bit, then when A is a 1, the result is the reverse of the other bit. When A is 0,

the result is the same as the other bit. To change a bit (to flip it) we XOR it with 1; to leave the bit as it is, we XOR it with 0.

Figure C.8 Truth Table for XOR

A	B	A XOR B
1	1	0
1	0	1
0	1	1
0	0	0

Though *PREG* has a truth table for XOR (XOR, we are told, is part of the WAIT statement), the operator does not appear in C64 BASIC. It seems a pity.

The XOR operator can be emulated by a combination of other operators. A XOR B is the same as

NOT(A AND B) AND (A OR B)

— at this point it is worth trying to work out why.

1) A AND B is only true when both A and B are true: NOT(A AND B) is only false when A and B are both true.
2) A OR B is only false when both A and B are false.
3) The only time when both NOT(A AND B) and A OR B are true is when A and B have different values: true is 1, and this is exactly what is shown in the XOR truth table.

To flip bit 5 of a value X, we XOR with 00100000 (= 32),

X = NOT(X AND 32) AND (X OR 32)

and if more bits were to be flipped, then the number would be that much more complicated — but easier if Appendix F is consulted.

How can we arrange our BASIC so that it does not conflict with our bit map? See Appendix D.

APPENDIX D
Re-arranging Memory

Try these commands, after typing NEW, or after switching on:

```
PRINT PEEK(43) + PEEK(44)*256
PRINT PEEK(45) + PEEK(46)*256
```

to which the response is 2049 and 2051 respectively.

Locations 43 and 44 are two bytes which together point to the start of the BASIC program: as there is no program, there should be nothing to follow. Or is there?

The other two locations (ie 45 and 46) point to the start of the BASIC variables, though at the moment there are none. I could have used other pointers (and for those who wish to check, they are on pages 311 and 312 of *PREG*). It seems as if the BASIC program occupies two bytes (ie 2051−2049) even though there is no program.

Try

```
PRINT PEEK(51) + PEEK(52)*256
PRINT PEEK(55) + PEEK(56)*256
```

which is, on both occasions, equal to 40960. Locations 51 and 52 point to the bottom of string storage (A$ is a string), whereas locations 55 and 56 point to the highest address used by BASIC. In C64 BASIC (and this is common to all Commodore BASICs), strings are stored at the top of memory, and slowly fill memory from the top downwards, until they meet the locations being filed from the bottom of memory upwards by the BASIC program variables, and arrays.

If we enter

```
PRINT FRE(0)
```

ie, PRINT out the amount of FREe memory, the result is −26627. We cannot have a negative amount of memory available, and, if we remember

the results about clock-like arithmetics in Appendix B, we might guess that we have to add 65536. So

```
PRINT FRE(0) + 65536
```

and we find the FRE memory is 38909.

If we subtract 38909 from 40960 the answer is 2051. FRE(0) gives the amount of memory free, between the top of the BASIC program and the highest address used by BASIC. Note that when you switch on the C64 you are told that 38911 BASIC bytes are free. This amount of free memory includes the BASIC program. (**Figure D.1** gives a list of useful locations: to find at which address they point, multiply the higher location by 256, as above.)

Figure D.1 BASIC Pointers

LOCATIONS	FUNCTION
43–44	Start of BASIC program
45–46	Start of BASIC variables
47–48	Start of BASIC arrays
49–50	End (+ 1) of BASIC arrays
51–52	Bottom of string storage
53–54	End of strings
55–56	Highest address used by BASIC
641–642	Bottom of memory for Operating System
643–644	Top of memory for Operating System

The arrangement of a BASIC program

It is now time to enter a BASIC program, but not until we have quickly studied the contents of the bytes from 2048 to 2051.

Remember we have no program in memory, and so in immediate mode (no line numbers) we enter

```
FOR I = 2048 TO 2051: PRINT I,PEEK(I): NEXT I
```

to which the response is (in my case)

```
2048    0
2049    0
2050    0
2051    73
```

and in your case (if you used a loop variable other than I) the 73 might well be different. The succession of three 0s is very important: these 0s are the way in which the end of a program is signalled to BASIC.

Now enter this program:

```
10 REM
20 PRINT "$$$"
```

and then (in immediate mode)

```
PRINT PEEK(43) + PEEK(44)*256
PRINT PEEK(45) + PEEK(46)*256
PRINT PEEK(51) + PEEK(52)*256
PRINT PEEK(55) + PEEK(56)*256
```

to which the responses are

```
2049
2068
40960
40960
```

and the only pointer to have a different value is that which points to the start of the BASIC variables. As the start of the BASIC variables signals the end of the program text, we know that we have a program in memory (though LISTing is simpler).

At the moment I do not want to have too great a confusion in the program, otherwise (instead of these double POKEs) we could have defined a DOUBLE POKE function

```
5 DEF FN DP(X) = PEEK(X) + PEEK(X + 1)*256
```

but we have to live with the more cumbersome double POKEs for the moment.

As we did before, we do now, with the only difference being that the start of the variables is at 2068:

```
FOR I = 2048 TO 2068: PRINT I,PEEK(I): NEXT I
```

and we produce

```
2048    0
2049    7
2050    8
2051    10
2052    0
2053    143
2054    0
2055    18
2056    8
2057    20
2058    0
2059    153
```

2060	34
2061	36
2062	36
2063	36
2064	34
2065	0
2066	0
2067	0
2068	73

The first byte (ie that at 2048, one less than the start of BASIC) is still 0, and will always have to be so. The next two bytes are pointers (again):

```
PRINT PEEK(2049) + PEEK(2050)*256
```

produces 2055 — which is a little further on from that location. Location 2055 is preceded by a location (2054!) which has a zero byte. If we

```
PRINT PEEK(2051) + PEEK(2052)*256
```

we find the answer 10. These two locations obviously contain the line number of the first line.

So far, therefore, we have established that the first two bytes point to the beginning of the next line, and the next two bytes contain the line number. Next, we would expect to find the word REM stored in some form, but we find that the value stored in location 2053 is actually 143. Turning to Appendix F of *MUM* (on ASCII codes) indicates that the code 143 is nothing to do with any of the letters of REM (see also my Appendix A).

The word REM is a C64 BASIC keyword, and — in common with other Commodore Microsoft BASICs — the word itself is not stored in memory. The BASIC interpreter stores a token to stand for that keyword. The token for REM is 143.

In Appendix D of *MUM*, shortened forms are given for many BASIC commands. In the introduction to these shortened forms it is noted that though the shortened form might have been used to enter the line, only the full version is listed. When the BASIC interpreter comes across a shortened form of a command, the command is still stored as a token. On listing, the token is recognized, and listed in full. (On this, more in a short while.)

Location 2054 is next, and it is a zero byte — indicating the end of the line. The bytes at locations 2049 and 2050 pointed to 2055, and this is the start of the new line of program text. The contents of the first two locations themselves point to another address, the address is

```
PRINT PEEK(2055) + PEEK(2056)*256
```

which produces the location 2066 (location 2066 contains a 0).

The two locations which follow contain the value of the line number, ie 20. After the line number we expect the first part of the statement, ie

PRINT: we find the value 153. 153 is the ASCII value of the token which corresponds to the command PRINT. The values of the tokens for keywords are given in **Figure D.2.**

Figure D.2 Keyword Tokens

TOKEN	COMMAND	TOKEN	COMMAND
128	END	129	FOR
130	NEXT	131	DATA
132	INPUT #	133	INPUT
134	DIM	135	READ
136	LET	137	GOTO
138	RUN	139	IF
140	RESTORE	141	GOSUB
142	RETURN	143	REM
144	STOP	145	ON
146	WAIT	147	LOAD
148	SAVE	149	VERIFY
150	DEF	151	POKE
152	PRINT #	153	PRINT
154	CONT	155	LIST
156	CLR	157	CMD
158	SYS	159	OPEN
160	CLOSE	161	GET
162	NEW	163	TAB(
164	TO	165	FN
166	SPC(	167	THEN
168	NOT	169	STEP
170	+	171	−
172	*	173	/
174		175	AND
176	OR	177	>
178	=	179	<
180	SGN	181	INT
182	ABS	183	USR
184	FRE	185	POS
186	SQR	187	RND
188	LOG	189	EXP
190	COS	191	SIN
192	TAN	193	ATN
194	PEEK	195	LEN
196	STR$	197	VAL
198	ASC	199	CHR$
200	LEFT$	201	RIGHT$
202	MID$		

Next we come to the value 34, and an examination of *MUM* (Appendix F) shows that the ASCII value 34 corresponds to the double quotes ". The three successive instances of 36 all correspond to $, and the following 36 is the closing quote. There is a 0 (location 2065) to indicate the end of a line.

The next two locations (ie 2066 and 2067) are where we would expect to find the address of the next line after that one. The address is 0, and so the system knows that the end of the program has been reached.

Immediately following location 2067 (ie the start of the BASIC variables) is the value 73, and if you

```
PRINT CHR$(I)
```

the C64 outputs the letter I. I was the name of the variable we used to count the loop in the immediate statement.

Memory matters

In the previous appendix, we effectively decided that the best place for the character memory (ie our bit map) was at 8192 to 16383. There is a slight problem, in that any reasonable size of BASIC program will soon eat into that patch of memory: funny and not so funny things might happen.

Ideally, therefore, we would like the BASIC program and variables to go up to 8192 and then jump automatically to 16384, leaving our bit map undisturbed. We cannot do that — or if we can, I do not know how to do it.

The simplest way around our problem is to arrange for BASIC to start somewhere else, preferably at 16384. Figure D.1 showed that the two bytes at locations 43 and 44 pointed to the start of BASIC. What happens if we alter them? Normally, location 44 contains the value 8, which is equivalent to 8*256 = 2048, and location 43 contains 1: they point to 2049.

To move the pointers up to 16384 (or, + 1, to 16385), we have to change the value stored in location 44 to 16384/256 = 64. To make sure all the other pointers are altered after we have modified location 44 (otherwise confusion will reign), we ask the system to reinitialize by typing NEW. Try it:

```
POKE 44,64 : NEW
```

and then possibly

```
LIST
```

— unless you are extremely fortunate there will then be a ?SYNTAX error somewhere along the way.

What we did was perfectly correct, apart from one small point. Ordinarily, BASIC starts at 2049 and is preceded by one byte which contains 0. We forgot to put a 0 in the location before the start of BASIC. BASIC starts at 16385, so in location 16384 (before we NEW) we POKE the value 0:

```
POKE 44,64 : POKE 16384,0 : NEW
```

This works, and can be checked by examining the other pointers at 46 and 48.

```
PRINT PEEK(46), PEEK(48)
```

This should, in both cases, give the answer 64. To raise the base of BASIC to 16K (ie 16384), when the top is at 40K (ie 40960), is still to leave 24K, which should be sufficient for most applications.

This is by far the simplest method of protecting character memory (and bit maps). There are, however, many other methods available. For example, it is possible to use the memory bank which starts at 32768, and lower the top of BASIC: this however is more complex, and we have to take more care about where we put the bit map. To lower the top of BASIC requires two POKEs (see Figure D.1).

```
POKE 56,X : POKE 52,X : NEW
```

where X determines where the new top is to be.

Using the simple technique of raising the start of BASIC, means that bit maps and user-defined characters can both be treated without any further adjustment to the siting of BASIC.

Characters are examined in Appendix E.

APPENDIX E
Character Memory

Though, in theory, it is very simple to create characters of one's own design on the C64, in practice — as always with the C64 — it requires rather more understanding of the hardware of the machine.

The 6510 microprocessor

The way in which characters are stored in the C64 seems rather strange at first. The original characters (ie those which appear on the screen) are stored as sets of bytes in the C64 BASIC ROM. The BASIC ROM is the source of the BASIC translator, the system which interprets your lines of BASIC program, and turns them into instructions the C64's microprocessor can understand.

Rom stands for Read Only Memory, and it is a petrified system: nothing you do can change what is in the ROM. Whereas we are able to change what is in RAM (Random Access Memory) by use of the POKE command, we can POKE in a ROM location, but the POKE has no effect. The standard character set is so petrified, and there is nothing we can do about it.

The C64 uses a MOS 6510 processor (an improved version of the 6502 processor used in many other computers), and one of the improvements to the 6510 is the way in which it uses memory. We have already seen how the VIC-II chip can be instructed to look at different portions of memory (the different video memory banks), and what can be performed by the 6510 is rather more complex than even that.

At location 1 in the C64 there is what is termed the 6510 input/output port. This port is a rather more sophisticated version of location 53272 (the VIC-II register which controls character memory, and similar items — (see Figure C.7). Its contents are shown in **Figure E.1**.

The BASIC ROM runs BASIC, and the KERNAL ROM runs the KERNAL: the KERNAL is the machine operating system. The KERNAL (on powering up the C64) takes the character set stored in the ROM at $D000 to $DFFF, and copies it to locations $1000 to $1FFF in RAM. This means that normally, until pointers are altered, the screen takes its information about the shape of characters from locations 4096–8191 — it thinks it does.

Depending upon whether switches are set or not the C64 can appear many different beasts to the 6510 processor: the C64 has a true 64K of RAM, and

depending upon these switches, some portions of RAM can be replaced by ROM systems. Normally, not all RAM is available because the system starts with the BASIC and KERNAL ROMS replacing RAM locations in the control of the processor.

Figure E.1 Location 1 — MOS 6510 On-chip Input/Output Port

BIT	FUNCTION
0	LORAM : 0 to switch BASIC ROM out (1 default)
1	HIRAM : 0 to switch KERNAL ROM out (1 default)
2	CHAREN : 0 to switch character ROM in, (1 default) switches in RAM
3	Cassette data output line
4	Cassette switch sense (1 switch closed)
5	Cassette motor control (0 ON, 1 OFF)
6–7	Undefined

Notes:

1. All bits (except bit 4, the cassette switch sense) are set for outputs
2. Bit 0 controls locations $A000 to $BFFF
3. Bit 1 controls locations $E000 to $FFFF
4. Bit 2 controls locations $D000 to $DFFF
5. The memory from $C000 to $CFFF is free from control, and it is suggested (eg *PREG* page 309) that this is the best place to put short machine code routines

BASIC and memory

The default situation for BASIC is at 2049 upwards (cf my Appendix D), and it will not take a great deal of programming for the BASIC program to overlap with the character set (the ROM image) stored from 4096 to 8095. From 2049 to 4096 is 2K, and a 2K program is a small program (eg about 50 lines of 40 characters, ignoring variables).

For any decent sized program there will be contamination — or is there? Your program or the variables, or both, now occupy the same space as your shapes should use. What actually happens is that the VIC-II chip thinks that it is looking at locations 4096 to 8195: however it is really looking at the shapes defined in ROM.

C64 BASIC treats characters in two different ways. When I enter

```
PRINT CHR$(147) : PRINT CHR$(65)
```

the screen clears, and an A appears one line down from the top, but when I enter

```
PRINT CHR$(147); : PRINT CHR$(65)
```

the screen clears, and the A appears on the top line.

The ASCII code for the upper-case A is 65 (see Appendix F of *MUM*), and when we print the character corresponding to the ASCII code 65 we print an A. (ASCII codes are fairly standard across computers).

Suppose, however, we

```
PRINT PEEK(1024)
```

to which the response is the number 1 (I am assuming that the A is still top left on the screen). Consulting page 63 of *MUM* indicates that location 1024 is the top left location on the screen. To PEEK at that location is thus to examine the value stored in memory corresponding to that screen position.

The C64 (as with all Commodore machines of this general type) stores the video picture by use of a different set of codes, rather than ASCII values. These are the screen display codes (Appendix E of *MUM*), and the screen display code for A is a 1.

The default arrangement of memory up to 40K, without any provision for a user-defined set of characters, is thus as shown in **Figure E.2**. We have to modify that arrangement to produce a situation where BASIC, and user-defined characters, can coexist.

Figure E.2 Memory to 40K — the Default Arrangement

LOCATIONS	FUNCTION
0–1023 ($0000–$03FF)	System RAM
1024–2047 ($0400–$07FF)	Screen Memory Area (video Matrix and Sprite pointers)
2048–4095 ($0800–$0FFF)	BASIC program space only
4096–8191 ($1000–$1FFF)	BASIC program space, and also character ROM image (do not conflict)
8192–40959 ($2000–$9FFF)	BASIC program space only

The solution is the same as before, raise the start of BASIC to 16384, and there are no problems. Raising the start of BASIC as a matter of course has much to recommend it, as a preventative measure. We raise the start, you will remember, by

```
POKE 44,64 : REM CHANGE THE "START OF BASIC" POINTER
POKE 16384,0 : REM START THE SEQUENCE WITH A 0
NEW
```

and it is as well to do that now.

Examining the new arrangement of memory to 40K (the memory map) reveals that there is now quite a reasonable amount of space available for machine code programs, which might be used in BASIC by the SYS command or USR function (*MUM* pages 125 and 127, and *PREG* Chapter 5).

Though using machine code might seem a long way off at the moment, the fact that we have already thought about the use of machine code — and made provision — means that at some time in the future we will not have to reorganize totally our memory arrangements.

If we assume that we will place character memory starting at 8196 and extending to 16383, which is also the place at which we will locate the bit map for high resolution graphics, then we produce the new arrangements of **Figure E.3**. This is an important figure and should be studied, and — possibly — remembered.

Figure E.3 Memory to 40K — the Modified Arrangement

LOCATIONS	FUNCTION
0–1023 ($0000–$03FF)	System RAM
1024–2047 ($0400–$07FF)	Screen Memory Area
2048–4095 ($0800–$0FFF)	Free
4096–8191 ($1000–$1FFF)	Free (and ROM character image)
8192–16383 ($2000–$3FFF)	Character memory or bit map
16384–40959 ($4000–$9FFF)	BASIC program space

Character memory

The largest extent of the character memory is 4K, and this means that, of the portion of memory set aside from 8K to 16K (ie 8192 to 16383), only half can be used for new characters. It's always possible, however, by setting the correct bits at location 53272 (see Appendix C, especially Figure C.7) to have two sets of new characters, and to switch between the two sets (of this, more later). This should not be confused with the two sets of video characters (*MUM* Appendix E).

The next question becomes: how are characters stored? *MUM* keeps mum about this (as with most things), and *PREG* has a short discussion on pages 107 to 114. The characters you see on the screen are composed of little blocks of foreground and background, where each character is eight little blocks across, and eight blocks down. There are 64(= 8x8) blocks, arranged in eight lines of eight.

Each little block can be called a pixel (and usually is so called), and so each character is composed of 64 pixels. Pixels also are important in bit map mode.

Examine the screen display character corresponding to code 79 for Set 1 graphics (*MUM* page 133). This character is shaped like a rotated L: to see it on the screen we can

```
POKE 1024,79
```

as well as

```
PRINT CHR$(111)
```

(the difference between the video code and the ASCII code — *MUM* page 136). There is a line along the lefthand side, and a line along the top. You may not have produced anything with the POKE — POKEing on the screen needs to have been activated by some other character being there in the first place. (As in Appendix A).

The shape of the character is given in detail in Figure E.4 where the * represents the foreground, and . shows the background. The eight foreground and background pixels going across each line are stored as the eight bits of a byte. The value stored for line 0 is 255.

Each activation of a pixel — ie the presence of foreground — is coded as a 1, and each dormant pixel (background) is coded as 0. The shape is thus able to be represented as the binary numbers in **Figure E.4**, where the binary numbers are calculated from first principles, or by reference to Appendix F.

Figure E.4 A Specimen Shape — Video 79, ASCII 111

The shape		*The values*	
		BINARY	DECIMAL
********	Line 0	11111111	255
********	Line 1	11111111	255
**.....	Line 2	11000000	192
**.....	Line 3	11000000	192
**.....	Line 4	11000000	192
**.....	Line 5	11000000	192
**.....	Line 6	11000000	192
**.....	Line 7	11000000	192

The character definitions in ROM are stored as numbers in eight successive locations, just as we have here. The video number (in this case 79) indicates how far the definition is along memory.

The character set in ROM, which starts at 53248, will have the first eight

bytes (ie 53248 + 0 to 53248 + 7) corresponding to the @ symbol (video code 0, see *MUM* Appendix E).

The same location (53248) is, incidentally, that which starts the VIC-II registers in RAM. To attempt to discover what is the pattern for @, by use of PEEKS, will only produce the coordinates of the first four sprites. (See *MUM* page 73, where V = 53248.)

To access the eight bytes corresponding to video code 79, we would need to start at 53248 + 79*8 + 0 and continue through to 53248 + 79*8 + 7: assuming, that is, that we switched out the RAM, and switched in the character ROM by altering location 1.

If there are 256 different characters to copy, and each takes up 8 bytes, we need to copy 256*8 bytes. We have to copy bytes from 53248 + 0 to 53248 + 256*8 – 1. This is what the first program accomplishes.

Loading characters

The program User-defined Shapes : Setting up the System is used to copy bytes from the ROM at 53248 onwards into memory at 8192 onwards. It would also be possible, afterwards, to copy from either 8192 or 53248 to a new start at 12288 (ie 12K). It would be simpler to copy from 8192 for reasons which will become obvious.

First of all we have to resituate BASIC at 16K by

```
POKE 44,64 : POKE 16384,0 : NEW
```

and then enter in the program either by hand or from cassette (or disk, even). There are several locations to note: 1 — mentioned earlier (Figure E.1); 53248 — the start of the ROM character set; 53272 — mentioned earlier (Figure C.7); and, 56334 — to be mentioned.

```
LIST
     1REM
     2REM ---------------------------------
     3REM
     4REM USER-DEFINED SHAPES : SETTING
     5REM UP THE SYSTEM
     6REM
     7REM ---------------------------------
     8REM
    90REM I M P O R T A N T   BASIC ALREA
DY RELOCATED BY POKE44,64:POKE16384,0:NE
W
    95REM
```

```
  100 PRINT CHR$(142) : REM SETS UPPER C
ASE FOR CHARACTERS
  110 POKE 56334, PEEK(56334) AND 254 :
REM TURNS OFF THE KEYBOARD
  120 POKE 1,PEEK(1) AND 251 : REM INSTR
UCT PROCESSOR TO CHOOSE CHARACTER SET
  130 FOR I=0 TO 8*256-1 : REM COPY ALL
THE CHARACTERS FROM ROM (IE 256)
  140 POKE I+8192,PEEK(I+53248) : REM CO
PY THE VALUE STORED IN ROM TO NEW SPACE
  150 NEXT I
  160 POKE 1, PEEK(1) OR 4 : REM SWITCH
TO USING MEMORY FOR VIC-II REGISTERS
  170 POKE 56334,PEEK(56334) OR 1 : REM
SWITCH ON KEYBOARD AGAIN
  180 POKE 53272,(PEEK(53272) AND 240) O
R 8 : REM POINT AT NEW CHARACTER SET
  999 END
>
```

Location 56334 is a CIA #1 Control Register (A), and it controls Timer A. Timer A is used in the scanning of various devices (the keyboard column, joystick, paddles, and light-pen, though not all at once).

If the clock is stopped, ie Timer A is stopped, the keyboard (if it is the preferred device) will not be scanned — it will not respond. If we want to disable the keyboard whilst some rather complex manipulations are underway, we do this by switching off the clock (Timer A). We switch off by forcing bit 0 of location 56334 to 0 (the default is 1).

Figure E.5 Location 56334 — CIA # Control Register A

BIT	FUNCTION
7	Time-of-Day Clock Frequency
6	Serial Port I/O Mode (1 is output)
5	Timer A Counts : 1 is CNT signals, 0 is System 02 clock
4	Force Load Timer A : 1 is yes
3	Timer A Run Mode : 1 is one-shot, 0 is continuous
2	Timer A Output Mode to PB6 : 1 is toggle, 0 is pulse
1	Timer A Output on PB6 : 1 is yes
0	Start/Stop Timer A : 1 is start

Note: only bit 0 is relevant.

Let us go steadily through the program, as we can learn a good deal of C64 philosophy from it.

To

```
PRINT CHR$(142)
```

is to produce upper case for characters (eg *MUM* page 136), though this does not matter terribly (it means we load Set 1 rather than Set 2).

When we POKE into location 56334 with AND 254, we set bit 0 to 0. This means that the keyboard then becomes inactive. The reason why we want to de-activate the keyboard is the POKE to location 1: with this POKE we set bit 2 to 0, and this means that we have switched in the Character ROM.

The VIC-II is disabled by this POKE, and the system does not recognize its existence (see Figure E.1): we have to switch in the ROM before we can perform our next action, which is to copy the contents of the ROM into RAM. The copying is accomplished by

```
POKE I + 8192,PEEK(I + 53248)
```

(as we noted above). We copy 256*8 = 2048 locations across (ie only half of the 4K of ROM for characters, ie only Set 1). 256 characters should suffice for most purposes.

Whilst the copying of memory is occurring the C64 is dead to the keyboard, and whatever you try to do it will not recognize any tampering. The only way to stop what is happening is to switch off. If the program was interrupted part way through, this would mean that everything would be confused: to regain normality would indeed be difficult. Disabling is a sensible precaution.

We now have a user-definable character set nicely lodged at locations 8192 + 0 to 8192 + 1023. We are still using the old ROM image which the VIC-II thinks is at 4096 to 8091. We change the location from which VIC considers the characters to come by modifying location 53272 (the VIC Control Register, Figure C.7). We alter the bits in the register so that the characters start from 8192.

If we used

```
POKE 53272, (PEEK(53272) AND 240) OR 8
```

before we had copied the characters, we would have garbage on the screen. There were no nice characters stored at 8192 onwards, just the state of memory at that time. To hold down STOP and RESTORE at the same time will automatically set VIC-II to its default way of looking at things.

After every restore, therefore, we need to enter the POKE into 53272 to reset the VIC-II to look at 8192.

The Character Designer

The subroutine User-defined Shapes : The Character Designer follows on from the previous program. It assumes that there is a character set stored from 8192 upwards, though (of course) there need not be. If there is no character set stored from that location, however, then the garbage on the screen is rather difficult to understand. The program still works.

```
LIST
 1000REM
 1010REM ------------------------------
 1020REM
 1030REM USER-DEFINED SHAPES : THE
 1040REM CHARACTER DESIGNER
 1050REM
 1060REM ------------------------------
 1070REM
 1080 PRINT CHR$(147) : REM CLEARS SCREE
N
 1090 INPUT "*VIDEO NUMBER "; CH : REM S
CREEN USES VIDEO NUMBERS, NOT ASCII
 1100 POKE 1064,CH : REM OVER-WRITES THE
 * WITH THE CHOSEN CHARACTER
 1110 FOR I=0 TO 7 : REM THE TOP LINE OF
 SHAPE IS CALLED 0
 1120 PRINT : PRINT "BINARY CODE ";I; :
INPUT A$: REM NUMBER INPUT AS STRING
 1130 IF LEN(A$) <> 8 THEN 1010 : REM TH
ERE HAVE TO BE 8 CHARACTERS IN A LINE
 1140 B=0 : FOR J=8 TO 1 STEP -1 : REM F
ORMS THE DECIMAL EQUIVALENT IN B
 1150 B=B+B : REM DOUBLE THE VALUE OF ST
ORED NUMBER
 1160 IF ASC(RIGHT$(A$,J)) = 49 THEN B=B
+1 : REM FORMS NUMBER FROM LEFT
 1170 NEXT J
 1180 POKE 8192+CH*8+I,B : REM VALUE IN
APPROPRIATE LOCATION
 1190 NEXT I : REM ON TO NEXT LINE
 1200 RETURN
>
```

Before the subroutine is used we need to make sure that VIC is looking at the correct place, so we use the immediate statements

```
POKE 53272,(PEEK(53272) AND 240) 8 : GOSUB 1000
```

and then progress with the subroutine.

The first line

```
PRINT CHR$(147)
```

clears the screen (*MUM* page 136) and moves to the second line. The next line outputs the query

```
*VIDEO NUMBER ?
```

to which the user has to provide the video number for the character to be redefined (eg @ is 0, *MUM* Appendix E). The asterisk preceding the word VIDEO is at the first position on the second line: equivalent to the location 1064 in memory.

The video number is POKEd into location 1064, and so over-writes the asterisk. This will provide a continuing display of our efforts as designers.

The segment from 1110 to 1190 is the part of the routine which changes the shape given to the character with that video number. It takes each line of the character at a time, starting with line 0 (those with *PREG* might like to compare page 112). For each line the routine asks for the binary code for that line (compare Figure E.4).

The binary code is input as a string (ie A$), and at 1130 there is a check to see that there are eight characters/bits in the string, otherwise the user is asked for that line again. The decimal number (to be POKEd later into the appropriate location) is formed in B. B is initialized to zero.

The string is investigated, character by character, from the leftmost bit to the right, and if the character is a 1 then the value of B is increased by 1. The check occurs by successively looking at smaller and smaller portions of the rightmost elements of A$ (ie RIGHT$(A$,J)).

The ASCII value of the leftmost element in the steadily decreased string is taken, and compared with 49. The ASCII code 49 is equivalent to the character 1. No check is made to see that the value is 48 (equivalent to 0), and so an improvement can be made there. As it stands the routine will accept anything instead of 1s for the 0s.

Consider the string

```
10101010
```

which is equivalent to a decimal number 170 (cf Appendix F). **Figure E.6** shows the successive stages in the calculation of B for this binary number. The segment from 1130 to 1170 is easily transported to other routines.

Figure E.6 The Decimal of 10101010

BIT	B+B	B+1?
Start	0	0
1 (7)	0	1
0 (6)	2	2
1 (5)	4	5
0 (4)	10	10
1 (3)	20	21
0 (2)	42	42
1 (1)	84	85
0 (0)	170	170

This is a very simple and effective method of performing such calculations, and can easily be extended to arithmetics of higher bases (all other bases are higher than 2).

When the number (B) has been calculated, then it is POKEd into the appropriate location in memory. As memory starts at 8192, and the locations are ordered in groups of 8 by the video number (with video numbers starting at 0), each character starts at 8192+8*CH (where CH is the video number).

Within each little patch of eight locations, the order is given by the line number. If the line within the character is I, then the appropriate location is 8192 + 8*CH + I, thus to POKE that location with the value B is to change that line in the character in the way we desired.

As we have the character always on display (just before VIDEO NUMBER ?) we can see the successive effects of the pattern we are creating.

More pixels

We can display 1000 characters on the screen at one time (ie a display of 40 columns and 25 rows). As each character has eight individually changeable pixels across, there are 40x8 = 320 separate pixel positions across the screen. Similarly there are 200 (ie 25x8) pixel positions down the screen.

On the screen as a whole, therefore, there are 320x200 = 64000 pixel positions on the screen at once. This resolution exactly corresponds to the high resolution graphics on the C64. When we come to discuss the arrangement of the bytes corresponding to a specific pixel position, we have to refer back continually to the arrangement of characters (mentally, at least).

We have always referred to the top row of characters as row 0, and the bottom row as row 24. Column/character 0 is on the left, and column/character 39 is on the right. Let us refer to pixels, therefore in a consistent manner: the Y coordinate of a pixel will go from 0 at the top to 199

at the bottom; anjd the X coordinate of a pixel will go from 0 at the left to 319 at the right.

If a pixel has a Y coordinate of YC then this pixel must lie within row INT(YC/8). That is, row 0 goes from Y coordinates 0 to 7, row 1 goes from Y coordinates 8 to 15, and so forth. The actual line of the row is given by subtracting eight times the row number from the coordinate.

If YC is 35, then the row number is INT(35/8) = 4: the line within that row is 35−4*8 = 3. That is, the pixels corresponding to that coordinate are in line 3 after 4 completed rows.

If the X coordinate is XC, then the column number, or the number of the character within a row is also INT(XC/8). The bit position within the character is not XC minus eight times the column position. The reason that this is not so (unlike the line within a row), is that bit 7 is to the left, and bit 0 is on the right. We subtract the pseudo bit position, just calculated, from 7.

We have

```
ROW = INT(YC/8)
LINE = YC-ROW*8
COLUMN = INT(XC/8)
BIT = 7-(XC-COLUMN*8)
```

and the location in memory corresponding to these coordinates is

```
LOCATION = 8192 + ROW*320 + COLUMN*8 + LINE
```

because there are 320 bytes per row, and 8 bytes per column (ie character), and a line is a line. To switch on a pixel, therefore, we need to activate the required bit of that location. To activate that bit we need to OR with 2^BIT.

```
POKE LOCATION, PEEK(LOCATION) OR 2^BIT
```

We use this information in Appendix G.

APPENDIX F
Binary Equivalents

The table in this appendix gives the binary and hexadecimal equivalents of decimal numbers from 0 to 255.

The table has been produced partly to aid those who want a quick method of looking up decimal values when designing shapes. For character definitions (Appendix E), my designer routines are perhaps more helpful.

In the case of the design of sprites they might perhaps have even greater utility. It is worth remembering (*MUM*, Chapter 6) that a sprite is three characters wide by almost (not quite) three characters deep.

A further reason for presenting the table is that it enables the reader to see clearly how not only binary, but also hexadecimal, relate to decimal values.

BINARY	HEX	DEC
00000000	0	0
00000001	1	1
00000010	2	2
00000011	3	3
00000100	4	4
00000101	5	5
00000110	6	6
00000111	7	7
00001000	8	8
00001001	9	9
00001010	A	10
00001011	B	11
00001100	C	12
00001101	D	13
00001110	E	14
00001111	F	15
00010000	10	16
00010001	11	17
00010010	12	18

00010011	13	19
00010100	14	20
00010101	15	21
00010110	16	22
00010111	17	23
00011000	18	24
00011001	19	25
00011010	1A	26
00011011	1B	27
00011100	1C	28
00011101	1D	29
00011110	1E	30
00011111	1F	31
00100000	20	32
00100001	21	33
00100010	22	34
00100011	23	35
00100100	24	36
00100101	25	37
00100110	26	38
00100111	27	39
00101000	28	40
00101001	29	41
00101010	2A	42
00101011	2B	43
00101100	2C	44
00101101	2D	45
00101110	2E	46
00101111	2F	47
00110000	30	48
00110001	31	49
00110010	32	50
00110011	33	51
00110100	34	52
00110101	35	53
00110110	36	54
00110111	37	55
00111000	38	56
00111001	39	57
00111010	3A	58
00111011	3B	59
00111100	3C	60
00111101	3D	61
00111110	3E	62

00111111	3F	63
01000000	40	64
01000001	41	65
01000010	42	66
01000011	43	67
01000100	44	68
01000101	45	69
01000110	46	70
01000111	47	71
01001000	48	72
01001001	49	73
01001010	4A	74
01001011	4B	75
01001100	4C	76
01001101	4D	77
01001110	4E	78
01001111	4F	79
01010000	50	80
01010001	51	81
01010010	52	82
01010011	53	83
01010100	54	84
01010101	55	85
01010110	56	86
01010111	57	87
01011000	58	88
01011001	59	89
01011010	5A	90
01011011	5B	91
01011100	5C	92
01011101	5D	93
01011110	5E	94
01011111	5F	95
01100000	60	96
01100001	61	97
01100010	62	98
01100011	63	99
01100100	64	100
01100101	65	101
01100110	66	102
01100111	67	103
01101000	68	104
01101001	69	105
01101010	6A	106

01101011	6B	107
01101100	6C	108
01101101	6D	109
01101110	6E	110
01101111	6F	111
01110000	70	112
01110001	71	113
01110010	72	114
01110011	73	115
01110100	74	116
01110101	75	117
01110110	76	118
01110111	77	119
01111000	78	120
01111001	79	121
01111010	7A	122
01111011	7B	123
01111100	7C	124
01111101	7D	125
01111110	7E	126
01111111	7F	127
10000000	80	128
10000001	81	129
10000010	82	130
10000011	83	131
10000100	84	132
10000101	85	133
10000110	86	134
10000111	87	135
10001000	88	136
10001001	89	137
10001010	8A	138
10001011	8B	139
10001100	8C	140
10001101	8D	141
10001110	8E	142
10001111	8F	143
10010000	90	144
10010001	91	145
10010010	92	146
10010011	93	147
10010100	94	148
10010101	95	149
10010110	96	150

10010111	97	151
10011000	98	152
10011001	99	153
10011010	9A	154
10011011	9B	155
10011100	9C	156
10011101	9D	157
10011110	9E	158
10011111	9F	159
10100000	A0	160
10100001	A1	161
10100010	A2	162
10100011	A3	163
10100100	A4	164
10100101	A5	165
10100110	A6	166
10100111	A7	167
10101000	A8	168
10101001	A9	169
10101010	AA	170
10101011	AB	171
10101100	AC	172
10101101	AD	173
10101110	AE	174
10101111	AF	175
10110000	B0	176
10110001	B1	177
10110010	B2	178
10110011	B3	179
10110100	B4	180
10110101	B5	181
10110110	B6	182
10110111	B7	183
10111000	B8	184
10111001	B9	185
10111010	BA	186
10111011	BB	187
10111100	BC	188
10111101	BD	189
10111110	BE	190
10111111	BF	191
11000000	C0	192
11000001	C1	193
11000010	C2	194

11000011	C3	195
11000100	C4	196
11000101	C5	197
11000110	C6	198
11000111	C7	199
11001000	C8	200
11001001	C9	201
11001010	CA	202
11001011	CB	203
11001100	CC	204
11001101	CD	205
11001110	CE	206
11001111	CF	207
11010000	D0	208
11010001	D1	209
11010010	D2	210
11010011	D3	211
11010100	D4	212
11010101	D5	213
11010110	D6	214
11010111	D7	215
11011000	D8	216
11011001	D9	217
11011010	DA	218
11011011	DB	219
11011100	DC	220
11011101	DD	221
11011110	DE	222
11011111	DF	223
11100000	E0	224
11100001	E1	225
11100010	E2	226
11100011	E3	227
11100100	E4	228
11100101	E5	229
11100110	E6	230
11100111	E7	231
11101000	E8	232
11101001	E9	233
11101010	EA	234
11101011	EB	235
11101100	EC	236
11101101	ED	237
11101110	EE	238

11101111	EF	239
11110000	F0	240
11110001	F1	241
11110010	F2	242
11110011	F3	243
11110100	F4	244
11110101	F5	245
11110110	F6	246
11110111	F7	247
11111000	F8	248
11111001	F9	249
11111010	FA	250
11111011	FB	251
11111100	FC	252
11111101	FD	253
11111110	FE	254
11111111	FF	255

>

APPENDIX G
Block Graphics

In the drawing of straight lines we will start with the drawing of lines in low resolution graphics: that is, the blocks we can put on the screen with ordinary PEEKs and POKEs.

The first line we draw will be from the top left hand corner at about 45 degrees (or at least from column 0, row 0, to column 24, row 24). The program is that called Lores Graphics: a Line of Blocks.

```
LIST
    10REM
    20REM ---------------------------------
    30REM
    40REM LORES GRAPHICS : A LINE OF BLOC
KS
    50REM
    60REM -------------------------------
    70REM
   100 PRINT CHR$(147) : REM CLEAR SCREEN
   110 POKE 53281,1 : REM SET BACKGROUND
TO WHITE
   120 POKE 53280,1 : REM SET BORDER TO W
HITE
   130 FOR Y=0 TO 24 : REM THERE ARE 25 R
OWS
   140 X=Y : REM SELECT COLUMN EQUAL TO R
OW
   150 POKE 1024 + X + Y*40,102 : REM THI
S IS THE MEMORY LOCATION, SHOW CHECKER
   160 NEXT Y
   170 GOTO 170 : REM DON'T SPOIL THE PIC
TURE
>
```

Low resolution graphics

The program is rather short, and when RUN turns the screen white all over (background and border). A line of checkered blocks (video code 102) then

runs diagonally down the screen to the right.

The program is rather simple. CHR$(147) clears the screen, and then the two POKEs set the background and border to the same colour — white (see *MUM* page 60). There is no good reason for this other than it looks rather effective when we draw the line.

As there are 40 columns (0 to 39) with 25 rows (0 to 24), and a simple line is to be drawn, we draw on each row (ignoring some columns). The row is indexed by the loop counter Y (from 0 to 24), and for each row we select the corresponding column. The column is indexed by the variable X, and X is made equal to Y.

At line 150 we plot the point, ie we POKE a value into the location corresponding to the intersection of the approporiate row and column (or the coordinates of the point). The value POKEd is 102, which is the video code corresponding to a checkered square (ie the checker).

When the line has been drawn, so as not to interfere with the display (the line is so pretty), we have the endless loop at line 170 (ie 170 GOTO 170). This is a fairly mundane line, and to stop the program press STOP. To get back to the proper initial state press STOP and RESTORE simultaneously.

Trying to produce something rather more adventurous results in Lores Graphics: Another Line of Blocks. The idea behind this program is to draw from the top left to bottom right corners. At column 0 we have row 0, and at column 39 we have row 24: each column is therefore worth about (is exactly) 25/40 rows. The gradient of the line is 25 in 40.

```
LIST
    10REM
    20REM ------------------------------
    30REM
    40REM LORES GRAPHICS : ANOTHER LINE O
F BLOCKS (?!)
    50REM
    60REM ------------------------------
    70REM
   100 PRINT CHR$(147) : POKE 53281,1 : P
OKE 53280,1
   110 S=25/40 : REM GRADIENT IS 25 ROWS
PER 40 COLUMNS
   120 FOR X=0 TO 39 : REM THERE ARE 40 C
OLUMNS
   130 Y=X*S : REM WORK OUT THE APPRORIAT
E ROW
   140 POKE 1024 + X + Y*40,102 : REM SEL
ECT MEMORY LOCATION, SHOW CHECKER
```

```
  150 NEXT X
  160 GOTO 160
>
```

Line 100 is effectively the same as lines 100–120 in the previous program. S stores the value of the gradient (ie 25/40 = .625). As there are more columns than rows, we have to step through all the columns, so that we do not miss any of the columns out — we guess that if we look after the columns, the rows will look after themselves.

If we plotted on each of the 25 rows, that would only be 25 points in total. There are 40 columns, and so 15 columns (at least) would have nothing plotted along their length. There would be gaps in the line.

We index the columns by the loop counter X (as before), and we then work out the value for the row which should correspond to that column. We call the row index Y. If the column is number 7 (say), the equivalent row is 7*S = 4.375. We plug these values of X and Y into the same formula as before (ie 1024 + X + Y*40).

The program is RUN. There is no line from the top left corner to the bottom right corner. There seems to be a regular pattern of dots all over the screen.

I made a mistake.

But where is the mistake? It must have something to do with the POKE to a location: perhaps we calculated the wrong location? The formula is the same as used previously, so the mistake must be in the way I calculated the Y coordinate value.

It is easy to remedy: we do not use the fractional value of Y in calculating the location, we use the value rounded to the nearest whole number. If Y is 4.375 then it is rounded to 4. The number Y = 4.375 means that the point lies somewhere on row 4, if the number was 4.75 this would be equivalent to row 5. So, we need to write a modified version.

```
LIST
    10REM
    20REM ------------------------------
    30REM
    40REM LORES GRAPHICS : THE MODIFIED A
NOTHER LINE OF BLOCKS
    50REM
    60REM ------------------------------
    70REM
   100 PRINT CHR$(147) : POKE 53281,1 : P
OKE 53280,1
   110 S=25/40 : REM GRADIENT IS 25 ROWS
PER 40 COLUMNS
```

```
  120 FOR X=0 TO 39 : REM THERE ARE 40 C
OLUMNS
  130 Y=X*S : REM WORK OUT THE APPRORIAT
E ROW
  140 POKE 1024 + X + INT(Y+.5)*40,102 :
 REM INTEGER MEMORY LOCATION, SHOW SHAPE
  150 NEXT X
  160 GOTO 160
>
```

The difference comes in line 140. Instead of

```
140 POKE 1024+X+Y*40,102
```

rather

```
140 POKE 1024+X+INT(Y+.5)*40,102
```

where INT(Y+.5) is the way in which we round to the nearest whole number. If the fractional part is .5 or over, adding .5 makes the number go over into the next whole number: INTing this new number then effectively rounds upwards.

If the fractional part is less than .5, the result of adding .5 still gives the same whole number. INT rounds down in this case. Running the modified version gives the straight line we want, diagonally, from corner to corner.

The Fan

Another example of low resolution graphics is the fan. We can see the effect of drawing more than one line (the same effect we get at higher resolutions) close up as it were.

We magnify the high resolution to see what happens.

The reason why I have started with low resolution lines is that high resolution lines are only low resolution lines, though rather finer. Or, alternatively, a low resolution line is nothing more than a high resolution line, but somewhat coarser.

What I want to draw is a series of lines. The lines all start at the same point (and the top left corner seems highly convenient). The lines then move to differing points along a side of the screen. The lines will go from top left to points down the right hand side of the screen.

Drawing the lines will involve different gradients. The line to the bottom right has a gradient of .625 (ie 24/40), and the line to the top right will have a gradient of 0. We want some way of drawing lines with a series of differing gradients.

If we think about the problem, we have the solution. In the previous program I used S to stand for the gradient, and so all we have to do is

arrange for the gradient to vary and the same drawing routine to be utilized, again and again.

```
   10REM
   20REM ------------------------------
   30REM
   40REM LORES GRAPHICS : THE PRETTY FAN
 SHAPE
   50REM
   60REM ------------------------------
   70REM
  100 PRINT CHR$(147) : POKE 53281,1 : P
OKE 53280,1
  110 FOR S=0 TO .625 STEP .1
  120 FOR X=0 TO 39
  130 Y=X*S
  140 POKE 1024 + X + INT(Y+.5)*40,102
  150 NEXT X
  160 NEXT S : REM MOVE TO NEW GRADIENT
  170 GOTO 170
>LIST
```

If you look at lines 120 to 150 in the Fan program, these lines are almost identical to the lines 120 to 150 in the Modified program (without REMarks). What has altered is that at line 110 we have a loop with values of S ranging from 0 to .625 in steps of .1, whereas in the earlier program S had only one value.

The program works, and it draws lines coming from the top left corner down the righthand side of the screen in small increments.

When drawing high resolution lines on the C64, or on almost any other computer, remember that those lines are no more than finer versions of the lines you have drawn here using the Fan program. Lines in high resolution are also as unmysterious as are these low resolution lines.

And low resolution lines are certainly highly unmysterious.

Low resolution coordinates

The final low resolution program appears to be rather more difficult, but it is not really all that complex. The complexity disappears once you know what is happening — but that is true of most things.

The purpose of the program is to draw lines between coordinates provided by the user.

First, what are coordinates? In the case of low resolution graphics, they are the column number and the row number in that order. The column

number is the X-coordinate, and the row number is the Y-coordinate. In the earlier programs, in fact, we used this convention.

Second, what are the limits on the coordinates? On the screen, the X-coordinate has a range from 0 to 39, and the Y-coordinate has a range from 0 to 24. Ideally, however, we would allow any coordinates to be used but we should only draw admissable values. We need some way of deciding how lines should be drawn if they go out of bounds.

The structure of the Lores Graphics: a Twixt Coordinates program is that the user is asked to stipulate beginning and end coordinates for two points. A line is drawn between these two points using one of two DRAW ROUTINEs. The routine used depends upon whether the separation between the coordinates is greater in the X direction or the Y direction.

We saw that when a fractional value of a row (or column) was used the plots were not accurate. In the case of this program, not only do we have to check that the intermediate values for coordinates are integers, but also we have to make sure that the beginning and end points are integers.

The lines we have drawn so far have all proceeded from lower values of X and Y to higher values (ie the line has always sloped downwards to the right). What if we used the initial coordinates 0,24 (bottom left) to 39,0 (top right)? The successive X values will increase as normal, and the correct Y values will be calculated and used.

What, however, if we start at 39,0 and go to 0,24? For starters: the loop would not work. If there is a loop

```
FOR X = 39 TO 0
```

it will only execute once (try it to check). To execute for all the values (assuming integral values) we have to write

```
FOR X = 39 TO 0 STEP -1
```

so that the BASIC system knows that we have to go down values in the loop. Note that 0−39 is negative, and so SGN(0−39) is equal to −1 (*MUM* page 127). The last version could thus be written

```
FOR X = 39 TO 0 STEP SGN(0-39)
```

or, more generally,

```
FOR X = LX TO NX STEP SGN(NX - LX)
```

Note that if LX equals NX, then the STEP is SGN(NX − LX) = 0: an eternal loop?

There are various ways of checking the appositeness of coordinates — we have to check them, otherwise we might POKE in all kinds of confusing places. The simplest method is first to calculate the location, and then — if the location is within correct bounds — POKE into that location.

This solution has some interesting consequences. A line drawn from 20,0

to 45,0 will proceed rightwards along the top row (ie row 0) from column 20. When the line reaches column 39, the next column is number 40, but there is no column 40.

The memory location corresponding to the top right corner on the screen is 1063 (ie 39,0), that corresponding to the coordinates 40,0 is one more, ie 1064. Memory location 1064 controls the character at 0,1 — column 0, row 1. The line wraps round the edge of the screen and appears at the left, one row down.

Wrap round can be fun.

To draw a line which goes off either the bottom of the screen or the top of the screen, is not to wrap round. It is possible to arrange wrap round at the top and bottom, but I will leave that as an exercise.

The program

```
   10REM
   20REM ------------------------------
   30REM
   40REM LORES GRAPHICS : A TWIXT COORDI
NATES
   50REM
   60REM ------------------------------
   70REM
  100 DEF FNCO(Z)= 1024 + X + Y*40 : REM
 CALCULATES LOCATION
  110 INPUT "INITIAL X AND Y ";LX,LY : R
EM START COORDINATES
  120 INPUT "FINAL X AND Y "; NX,LY : RE
M DRAW TO THESE
  130 IF LX-NX = 0 AND LY-NY = 0 THEN EN
D : REM NOWHERE
  140 PRINT CHR$(147)
  150 POKE 53281,1 : REM ONLY BACKGROUND
 CHANGED, I WANT TO SEE EDGES
  160 IF ABS(LX-NX) >= ABS(LY-NY) THEN G
OSUB 1000 : REM WIDER ON X AXIS
  170 IF ABS(LX-NX) < ABS(LY-NY) THEN GO
SUB 2000 : REM WIDER ON Y
  180 GOTO 180
  200 END
  980REM
  985REM DRAW ROUTINE 1
  990REM
 1000 S=(LY-NY)/(LX-NX) : REM GRADIENT
```

```
 1010 FOR X=INT(LX+.5) TO INT(NX+.5) STE
P SGN(NX-LX) : REM VARY X COORDINATE
 1020 Y=INT((X-LX)*S+LY+.5) : REM INTEGE
R Y COORDINATE
 1030 P=FNCO(0) : REM CALCULATE POSITION
 (DUMMY PARAMETER TO FUNCTION)
 1040 IF P>1023 AND P<2025 THEN POKE P,1
02 : REM ONLY POKE IF WITHIN BOUNDS
 1050 NEXT X
 1060 RETURN
 1980REM
 1985REM DRAW ROUTINE 2
 1990REM
 2000 S=(LX-NX)/(LY-NY)
 2010 FOR Y=INT(LY+.5) TO INT (NY+.5) ST
EP SGN(NY-LY)
 2020 X=INT((Y-LY)*S+LX+.5)
 2030 P=FNCO(0)
 2040 IF P>1023 AND P<2025 THEN POKE P,1
02
 2050 NEXT Y
 2060 RETURN
>
```

One of the most common needs for a plotting program is the calculation of memory locations which correspond to screen coordinates.

This calculation is made into a function FNCO(Z) in line 100 of the program. The Z is not used in the function itself. The Z is a dummy, and it is only there because the BASIC translator expects there to be a parameter for every function. (See *MUM* page 126.)

The function assumes that the X and Y coordinates (to which it refers in the calculation) are already in integral form. This is one aspect which we will have to remember in the writing of the program, though we could have used a slightly different definition

DEF FNCO(Z) = 1024 + INT(X + .5) + INT(Y + .5)*40

which would be more general.

The function is at the beginning of the program because, in that way, the BASIC translator will always have met the function before we use it. If the translator has not already encountered the function, we produce (*MUM* page 151) an ?UNDEF'D FUNCTION error.

The initial coordinate values for the line will be called LX and LY (I will use these names for my high resolution graphics, to illustrate the

similarities). The final coordinates will be called NX and NY. Lines 110 and 120 ask for the ends of the line.

If LX – NX is zero and also LY – NY is zero (line 130) then the program ends. If both differences are zero, then we are being asked to plot a point. I have decided that I do not want to plot points, though I could quite easily by altering this line to

```
155 IF LX - NX = 0 AND LY - NY = 0 THEN GOSUB 3000
```

and adding

```
3000 X = INT(LX + .5) : Y = INT(LY + .5) : P = FNCO(0)
3010 IF P> 1023 AND P< 2025 THEN POKE P,102
3020 GOTO 3020
3030 RETURN
```

(the astute reader will realize that line 3030 is never reached).

Line 130 will have to be omitted in the newer version, to be replaced by line 155.

The screen is cleared and the background (only) is set to white. I have not altered the border colour, because it is then clearer where the limits to the screen are. Having borders which are visible assists in the investigation of phenomena such as wrap round (wrap round appears above). If you want to change the border colour to another, you know what to do.

We have to decide whether we draw in steps along the X axis or the Y axis. In other words, are the ends of the line separated by more columns than rows, or vice versa?

Line 160 checks to see if the difference between the X coordinates is greater than the difference between the Y coordinates. Notice that the check is > = and not just > . This is because it is possible for the two differences to be the same, and we have to take that into account.

The reverse check is that in line 170, where the reverse of > = is < . The reverse of < is not > , and this is where many programmers make mistakes. The equality is ignored at one's peril.

These two lines refer to two almost identical routines, those at 1000 and 2000 (for example, lines 1030 and 1040 are identical to lines 2030 and 2040). The difference comes from the axis which is used for the loop mechanism.

Subroutine 1000 uses the X axis as given and calculates the Y values from that information, whereas the subroutine at 2000 uses the Y axis as given. If we understand subroutine 1000, then we understand subroutine 2000.

For each routine we first calculate the gradient. The formula for the gradient depends upon which axis we are taking as fixed. When the X axis is fixed (the subroutine at 1000) the gradient is the difference in the Y coordinates divided by the X difference. Subroutine 2000 fixes the Y values, and so the gradient is the X difference divided by the Y difference.

Remember the gradient: another name for it is the tangent (shown by

TAN(Z), *MUM* page 127).

We step from our initial coordinates to the final coordinates (where the coordinates are rounded to the nearest whole number). The direction in which we STEP depends upon the SiGN of the difference between the coordinates (lines 1010, 2010). I will call the coordinates, given by the loop counter, the fixed coordinates.

The other coordinate, corresponding to the fixed coordinate, is calculated by use of the gradient, and rounded (1020, 2020). This is the derived coordinate.

The fixed coordinate and the derived coordinate are then used (1030, 2030) to calculate the appropriate location, by use of FNCO(0). The 0 as parameter to FNCO is irrelevant, because any number or variable could be used, as it makes no difference. The BASIC function FRE(Z) (*MUM* page 129) also has a dummy parameter.

The value 102 is then POKEd into that location, if the location is within bounds: the checker appears (lines 1040, 2040). It would be possible to subroutine lines 1030–1040, 2030–2040, but I thought that spelling them both out makes the workings somewhat clearer.

Trying out this program for many differing values of the initial and final coordinates is an exercise well worth the effort. It is by trying out such lines that the high resolution routines (in the next appendix) become more easily understood.

To high resolution.

APPENDIX H
Pixel Graphics

The only difference between high resolution graphics and the equivalent version in low resolution is the size of the points plotted.

In low resolution we were plotting with a maximum of 40x25 points, whereas in high resolution we plot with a maximum of 320x200 points. The points in low resolution are characters and the points in high resolution are pixels.

At this point it might be worth referring to Appendix E, on Character Memory, and — in particular — the calculation of pixel positions. The most difficult transition to be made from low resolution to high resolution (as far as programming is concerned) is the manipulation of bits corresponding to coordinates.

The bit map

We have already met the bit map and location 53265 in Appendix C (especially consult Figure C.5). To switch into bit map mode we set bit 5 of location 53265 to 1. This has to be our second action on preparing to use high resolution graphics.

The first action is

```
POKE 44,64 : POKE 16384,0 : NEW
```

as always.

To switch bit 5 of a location to a 1, without altering the values of the other bits, we have to OR with 2^5, ie 32. This is the POKE we use:

```
POKE 53265,PEEK(53265) OR 32
```

however, we also use another POKE to that location. The other instruction is

```
POKE 53265,PEEK(53265) AND 239
```

which (as 16 = 255 – 239, and 16 = 2^4) means that bit 4 of location 53265 has been set to 0.

Examining Figure C.5 reveals that bit 4 controls the screen blanking. If bit 4 is set to 0, then the screen background becomes the same as the border, and the program runs rather more quickly. When the C64 is loading from cassette, for example, the screen goes blank and bit 4 of location 53265 has been set to 0.

While we are playing silly devils with memory (POKEing 0s into the 8K from 8192 to 16191 — well almost 8K), to speed up matters we can set bit 4 of location 53265. When we copied characters into memory, we might also have used the same facility.

Another important location is (as you have already guessed) our old friend 53272 (Figure C.7, and many other places). We use

```
POKE 53272,(PEEK(53272) AND 240) OR 8
```

to inform VIC-II which spot of memory to consider as containing the bit map. At first the bit map may contain rubbish, and so we have to clear out that patch of memory. We initialize by POKEing a 0 into every location.

The colour for the bit map comes now from the screen memory (not colour memory). The way in which the colour information is stored is rather clever. If you refer to *MUM* (page 61) you will see colour codes which range from 0 to 15. Each of those numbers could easily be stored in four bits. The lower four bits in a location in screen memory set the colour of the high resolution graphics background. The upper four bits give the colour of the plotted pixels.

Within the bit map memory (not the screen memory) the system recognizes background when the bit corresponding to that pixel is equal to 0. When the bit is set to 1, then that pixel has been plotted. The pixel can be considered to reside within a screen character, and so the system takes that pixel (and its bit value), examining the numbers stored in the equivalent location of screen memory.

If the number stored in the equivalent screen location is 1 (ie 00000001 as a binary number) the background is number 1 (white) and the foreground is 0 (black). We have to set up screen memory before we use high resolution graphics, by POKEing the same value in the locations from 1024 to 2023.

All these housekeeping tasks are performed by the Hires Graphics : Initialization subroutine.

```
1000REM
1010REM ------------------------------------
1020REM
1030REM HIRES GRAPHICS : INITIALIZATION
1040REM
1050REM ------------------------------------
1060REM
```

```
10000 GOSUB 11000 : REM ACTIVATE FUNCTIO
NS
10010 POKE 53265,PEEK(53265) AND 239 : R
EM "TURN OFF" SCREEN
10020 POKE 53265,PEEK(53265) OR 32 : REM
 ENABLE BIT MAP MODE
10030 POKE 53272,(PEEK(53272) AND 240) OR
 8 : REM POINT VIC-II
10040 FOR I=8192 TO 16191
10050 POKE I,0 : REM CLEAR BIT MAP
10060 NEXT I
10070 FOR I=1024 TO 1023
10080 POKE I,1 : REM WHITE BACK, BLACK P
LOT
10090 NEXT I
10100 POKE 53265,PEEK(53265) OR 16 : REM
 GET THE SCREEN BACK
10110 RETURN
```

The Initialization routine starts with something we have not mentioned, a call to a subroutine which contains all the functions to be used. The functions are placed in a subroutine so that when new functions are added they can merely extend the routine. The call to subroutine 11000 will always, therefore, reference all the functions.

As long as we initialize (and we will) we have initialized all the functions.

We switch off the screen (blank it to the border) by the POKE to 53265 with AND 239, then activate the bit map and inform VIC-II where to look — the POKEs to locations 53265 and 53272. It is as well to remind you once again that the start of BASIC has to be raised, otherwise funny things might happen. So remember to POKE to locations 44 and 16384.

The bit map memory is cleared (by POKEing in 0), and then the screen memory is used to set the graphics colours. The choice of white background and black plotting might not be your taste. If, for example, you would prefer a black background, and white plotting the value is 00010000 (in binary). This means that the number 16 is POKEd into the screen memory locations. The last action is to give us back the screen.

Functions

When I discussed character memory (Appendix E) I examined briefly how to locate a byte in memory corresponding to a pair of coordinates. The functions, at 11000, attempt to make that calculation simpler.

```
LISTTLIST
 1000REM
 1010REM ------------------------------
 1020REM
 1030REM HIRES GRAPHICS : FUNCTIONS
 1040REM
 1050REM ------------------------------
 1060REM
11000 DEF FNCO(Z) = 8192 + 320*FNCH(Y) +
 8*FNCH(X) + Y - 8*FNCH(Y)
11010 DEF FNCH(Z) = INT(Z/8) : REM ROW A
ND COLUMN NUMBER
11020 DEF FNBI(Z) = 7 + FNCH(Z)*8 - Z :
REM BIT POSITION IN BYTE GIVEN BY FNCO
11030 RETURN
>
```

Memory starts at location 8192, and there are 320 bytes per row, with 8 bytes at each character position/low resolution column. The 8 bytes are indexed by the finer (high resolution) coordinate Y, and how far through the low resolution row that coordinate is. We might say that Y measures the number of high resolution lines of bytes we are through the screen.

If the Y coordinate is 157, there are 19 completed low resolution rows, and that value of Y is located 157 – 8*19 = 5 high resolution lines through that low resolution row. (The first line is 0 lines through the row).

FNCO(Z) calculates that location for coordinates X and Y. We start at memory location 8192. FNCH(Y) gives the number of the low resolution row which contains the coordinate Y. The row number is then multiplied by 320 to determine the number of locations before the present row.

FNCH(X) then gives the number of the low resolution column which contains the coordinate X. This column is then multiplied by 8 to determine the number of bytes prior to that column (in that row). Compare the calculation of coordinates for low resolution graphics, because they are similar, but instead of 320 we have 40, and for 8 we have 1.

If the number of high resolution lines in the completed rows is subtracted from the Y coordinate value, then we have the number of lines through that low resolution row. This explains Y – 8*FNCH(Y). We have thus calculated the location.

The DEFined FuNction FNCH(Z) is simply a division by 8 and an INTegering: this function gives the low resolution character position.

Just as we calculated the number of lines through a row, so we have to calculate the bit position through a byte. Unfortunately, however, the bits are numbered in a reverse order to the low resolution columns — the columns go from 0 to 39, but the bits go from 7 to 0. The calculation is exactly

the same as that for the lines, except that we then subtract from 7. This is performed by FNBI(Z).

Omitting the GOSUB 11000 from the initialization produces an error. This is an important subroutine, and will be extended for turtle graphics.

Drawing lines

One of the important decisions to make in drawing lines with low resolution graphics is the choice between whether to fix the X coordinates or the Y coordinates.

```
LIST
 1000REM
 1010REM ------------------------------
 1020REM
 1030REM HIRES GRAPHICS : LINE CHOICE
 1040REM
 1050REM ------------------------------
 1060REM
12000 IF ABS(LX-NX) >= ABS(LY-NY) THEN G
OSUB 14000 : REM X IS FIXED
12010 IF ABS(LX-NX) < ABS(LY-NY) THEN GO
SUB 15000 : REM Y IS FIXED
12020 RETURN
>
```

The routine Line Choice is no different from its equivalent in low resolution graphics — why should it be different, the method is the same? Line Choice then refers to two other routines at 14000 to 15000. In drawing a line we use GOSUB 12000 (ie a call to this subroutine).

This routine is unexceptional.

```
LIST
 1000REM
 1010REM ------------------------------
 1020REM
 1030REM HIRES GRAPHICS : X COORD FIXED
 1040REM
 1050REM ------------------------------
 1060REM
14000 S=0 : IF LX-NX <> 0 THEN S=(LY-NY)
/(LX-NX) : REM GRADIENT
14010 FOR X=INT(LX+.5) TO INT(NX+.5) STE
P SGN(NX-LX) : REM FIX X
14020 Y=INT((X-LX)*S+.5+LY):REM DERIVE Y
```

```
14030 P=FNCO(0) : IF P > 8191 AND P < 16
192 THEN POKE P,PEEK(P) OR 2^FNBI(X)
14035 REM THE ABOVE POKES A 1 INTO BIT F
NBI(X) AT LOCATION P, IF P ADMISSABLE
14040 NEXT X
14050 RETURN
>
```

```
LIST
 1000REM
 1010REM ------------------------------
 1020REM
 1030REM HIRES GRAPHICS : Y COORD FIXED
 1040REM
 1050REM ------------------------------
 1060REM
15000 S=0 : IF LY-NY <> 0 THEN S=(LX-NX)
/(LY-NY)
15010 FOR Y=INT(LY+.5) TO INT(NY+.5) STE
P SGN(NY-LY)
15020 X=INT((Y-LY)*S+.5+LX)
15030 P=FNCO(0) : IF P > 8191 AND P < 16
192 THEN POKE P,PEEK(P) OR 2^FNBI(X)
15040 NEXT Y
15050 RETURN
>
```

The two routines at 14000 and 15000 are almost identical. They perform exactly the same duties, but one has rather more regard for the X axis, and the other prefers the Y axis. To examine one routine is to examine them both.

Subroutine 14000 (which is never needed explicitly by the user, who only needs 12000) starts with setting the gradient (S) to 0. As we are concentrating on the X axis, we find whether LX = NX, and, if not, we calculate the proper gradient. If there was no difference in value between LX and NX, when we divided by (LX – NX) we would divide by zero: not a good idea.

From there on, the routine differs little from the equivalent low resolution routine. The little is in the POKEing.

To set a specified BIT number to 1 we OR with 2^BIT, as we did above when we set bit 5 of location 53265 to 1, by OR 32. FNBI(X) gives the bit position corresponding to the high resolution coordinate X, so we OR 2^FNBI(X).

The workhorse of the set of routines is GOSUB 12000 (incidentally have you noticed how so many books about the C64 keep referring to workhorses?)

Demonstration Lines

There are two excessively long demonstration programs to keep you occupied long into the winter nights. The first is called (surprisingly) DEMO 1.

```
LIST
    10REM
    20REM ------------------------------------
    30REM
    40REM HIRES GRAPHICS : DEMO 1
    50REM
    60REM ------------------------------------
    70REM
    80 INPUT"LX,LY ";LX,LY : INPUT"NX,NY
";NX,NY : REM GET THE COORDINATES
    90 GOSUB 10000 : REM INITIALIZE
   100 GOSUB 12000 : REM DRAW LINE
   110 GOTO 110
   120 END
>
```

This program merely asks for your starting, and finishing, coordinates, initializes the high resolution system, and then draws a line, ending with an endless GOTO. This is fine for experimenting with reasonable and unreasonable coordinates, to see what occurs.

This routine is not much different in purpose to the equivalent low resolution routine, and it works on the same principles.

The next demonstration, DEMO 2, is a curve drawing routine. The turtle graphics routines use the same principles, and this routine might explain why the coordinates are named LX,LY and NX,NY.

```
10REM
20REM ------------------------------------
30REM
40REM HIRES GRAPHICS : DEMO 2
50REM
60REM ------------------------------------
70REM
80 GOSUB 10000
90 LX=0 : LY=0 : REM THE LAST COORDINATE
```

```
S
100 FOR NX=10 TO 300 STEP 20 : REM ALTER
 THE NEXT X VALUES
110 NY=NX*NX/300 : REM SET THE NEW Y VAL
UE WITH REFERENCE TO NEW X
120 GOSUB 12000 : REM DRAWS FROM LAST CO
ORDINATES TO NEW COORDINATES
130 LX=NX : LY=NY : REM THE NEW COORDINA
TES BECOME THE LAST COORDINATES
140 NEXT NX
150 GOTO 150
160 END
```

We initialize. The starting values for X and Y (ie LX and LY) are made equal to 0. We step along the X axis in units of 20, from 0 to 300. We call the next X value at each point NX, and we calculate the next Y value (ie NY) by the quadratic formula

NY = NX*NX/300

(which is equivalent to the arithmetical y = x*x/300). When the next/new values of X and Y have been calculated, we call GOSUB 12000. This routine, as we know, will draw a line from LX,LY to NX,NY.

A line is drawn then from the original/last value of X and Y to the new value of X and Y. When we have drawn that line, we make the last value of X and Y equal to the new values. As the loop moves on we are provided with a changed pair of new values.

What happens, then, is that a set of straight lines is drawn following a parabolic curve, until the line disappears off the bottom of the screen. Though the parabola is composed of straight lines, it still appears curved.

Understanding this program, and trying new functions at line 110, will help you to get a good grasp of the unseen working of the turtle graphics routines.

APPENDIX I
Variegated Graphics

An obvious extension to two-colour high resolution graphics is to use more than two colours. The C64 has provision for such an extension, to three foreground colours per character block, plus the background colour.

There is a slight discussion of this in *PREG* (pages 127–128) and there is also a short discussion of multicoloured characters (*PREG* pages 115–119). Essentially, however, multicoloured high resolution graphics are uncharted territory.

```
*WARNING*
REMEMBER BEFORE ALL ELSE
POKE 44,64 : POKE 16384,0 : NEW
```

The character block

In high resolution graphics we are concerned with bits in an 8x8 character block (where there are effectively 1000 such blocks). When we use high resolution graphics, we are working towards this type of analysis

```
..*.....
..*.....
...*....
...*....
....*...
....*...
.....*..
.....*..
```

which — under magnification — is what a straight line might look like as actually drawn.

Each bit (or pixel within the above character block) can either be 0 (ie '.') or 1 (ie '*'). Two colours can be used, depending upon the value of the bit. The two colours for the block are set by screen memory for the corresponding character in low resolution graphics.

In multicolour mode (which is set by forcing bit 4 at location 53270 to a 1, ie OR 16), the above line would look like

```
..**....
..**....
..**....
..**....
....**..
....**..
....**..
....**..
```

that is, the resolution is halved: the line is two pixels (ie two bits) wide. As the resolution is now less, the lines drawn are rather cruder approximations to straight lines.

As the number of pixels has doubled, the number of colours with which the system potentially can cope is now four, instead of two, colours. Whereas previously we had two possible values (ie 0 and 1) we now have four possible values, and each value takes its colour cue from a different place.

In **Figure I.1** I give the possible bit patterns, and their corresponding colour cues. It can be seen that the setting up of a multicolour high resolution system is slightly more complex than the setting up of a two-colour system.

Figure I.1

BIT PATTERN	COLOUR CUE
00	Background colour #0
01	Upper four bits of screen memory
10	Lower four bits of screen memory
11	Lower four bits of colour memory

The background colour #0 is that which is set by using location 53281. This is the normal location one uses to set the colour of the background (see page 60 of *MUM*), and was used in my Appendix A.

Screen memory is that portion of memory from 1024 to 2023 which is normally used to store the video number of a character at that corresponding location on the screen (again, see Appendix A). The eight bits at each location in screen memory can store 256 different values, and — as there are only sixteen different values — we can store two different sets of colours. Each set takes up four bits.

The fourth colour is stored at the location in colour memory which corresponds to the character block in which the bits are set. Only the lower four bits (a 'nybble' or 'nibble') of colour memory are used.

Note that the background colour used is the same throughout the screen,

but — in theory — the foreground colours (the lines) can vary from one character block to another. The point of this continuous variation may not appear obvious.

Initialization

```
10REM
20REM ------------------------------
30REM
40REM MULTI-COLOUR HIRES : INITIALISE
50REM
60REM ------------------------------
70REM
10000 PRINT CHR$(147) : REM CLEAR SCREEN
10010 GOSUB 11000 : REM ACTIVATE FUNCTIONS
10020 POKE 53265,PEEK(53265) AND 239 : REM
BLANK SCRREN
10030 POKE 53265, PEEK(53265) OR 32 : REM BIT
MAP MODE
10040 POKE 53272,(PEEK(53272) AND 240) OR 8 :
REM LOCATE CHARACTER MEMORY
10050 POKE 53270,PEEK(53270) OR 16 : REM
TURN ON MULTI-COLOUR MODE
10060 FOR I=8192 TO 16192
10070 POKE I,0 : REM CLEAR THIS PATCH OF
MEMORY
10080 NEXT I
10090 FOR I=1024 TO 2023
10100 POKE I,65 :REM COLOUR 1 IS RED, C
OLOUR 2 IS WHITE
10110 POKE 54272+I,6 : REM COLOUR 3 IS
BLUE
10120 NEXT I
10130 POKE 53281,7 : REM BACKGROUND IS
YELLOW
10140 POKE 53265,PEEK(53265) OR 16 : REM
SCREEN BACK ON
10150 RETURN
```

The easiest way for us now to see how to use the multicolour potential is for us to use it.

The multicolour high resolution system is initialized by use of the subroutine at line 10000, and it is worth comparing this to the earlier subroutine 10000 (for two-colour high resolution graphics).

Subroutine 10000 starts more or less like the other version. The screen is cleared (PRINT CHR$(147)) and the functions are activated. We blank the screen, as before, by POKEing into location 53265 (though some people actually prefer it unblanked, to watch what little is happening).

The same location is used again, this time to switch to bit map mode, and at the same time we direct the VIC-II to look at 8192 onwards by POKEing into 53272 (line 10040).

The important new POKE is that at line 10050, where we set bit 4 of location 53270 to be on (ie 1) by

```
POKE 53270,PEEK(53270) OR 16
```

this switches into multicolour bit map mode. Actually switching that bit on or off, whilst set up for multicolour, can give some amusing effects.

We clear the same patch of memory by the same method as before, that is, we POKE the value 0 into every location. By POKEing 0 into every location we set all the bits to 0, and — as can be gathered from Figure I.1 — this sets all pixels to the background colour.

When the bit map memory has been cleared, we initialize colours 01, 10, and 11.

First we initialize colours 01 and 10. The colour corresponding to the bit pattern 01 is given by the upper four bits of the corresponding location in screen memory: the colour corresponding to the bit pattern 10 is given by the lower four bits of the same location in screen memory.

Into the screen memory locations we POKE the value 65 (at line 10100), which we regard as being in two parts. As a binary number, 65 is 00100001, which on being divided into two nybbles of four bits gives 0010 0001. These are the numbers 2 and 1.

Referring to *MUM* (page 61) indicates that Colour 1 is RED (which has a colour code of 2 = 0010) and that Colour 2 is WHITE (with a colour code of 1 = 0001). Pixels can thus be red or white when active, plus one other colour. The other colour is that given by colour memory (ie corresponding to a bit pattern of 11, see Figure I.1).

The colour set by colour memory (ie line 10110) is 6, which corresponds to the colour code for BLUE. So, our foreground pixels can be red, white, or blue.

The background on which these colours are drawn is that given by the POKE to location 53281 (in line 10130). This POKE sets the background colour to YELLOW (corresponding to a colour code of 7).

Finally, we switch on the screen to start work — on the drawing of lines.

Drawing

As with the ordinary two-colour high resolution system, there is only one drawing routine — as far as the user is concerned. To draw a line is simply to

make the call GOSUB 12000 — as it was before.

```
10REM
20REM ------------------------------------
30REM
40REM MULTI-COLOUR HIRES : LINE CHOICE
50REM
60REM ------------------------------------
70REM
12000 IF ABS(LX-NX) >= ABS(LY-NY) THEN G
OSUB 14000 : REM X FIXED
12010 IF ABS(LX-NX) = ABS(LY-NY) THEN GO
SUB 15000 : REM Y FIXED
12020 RETURN
```

The content of the subroutine at 12000 is no different from that of the routine for the two-colour high resolution system of the previous appendix, and there is little difference in the drawing routines called at 14000 and at 15000.

Before we discuss the content of subroutine 14000 (remembering that subroutine 15000 is almost identical), first we will see where the functions differ.

The functions are collected in subroutine 11000, and the only one to differ is FNBI(Z). As we are dealing with pairs of bits at a time (ie 0,1 ; 2,3 ; 4,5 ; and 6,7) we do not wish to change an individual bit. The scaling of the X and Y axes has not been altered (the coordinates are not square as it is), and so all the other functions are still operative.

```
10REM
20REM ------------------------------------
30REM
40REM MULTI-COLOUR HIRES : FUNCTIONS
50REM
60REM ------------------------------------
70REM
11000 DEF FNCO (Z) = 8192+320*FNCH(Y)+8*FNCH
(X) +Y-8*FNCH(Y) : REM CALC LOCATIONS
11010 DEF FNCH(Z)=INT(Z/8)
11020 DEF FNBI(Z)=INT((7+FNCH(Z)*8-Z)/2)
: REM PAIR OF BITS START HERE
11030 RETURN
```

```
10REM
20REM ------------------------------------
30REM
40REM MULTI-COLOUR HIRES : X FIXED
50REM
60REM ------------------------------------
70REM
14000 S=0 : IF LX-NX <> 0 THEN S=(LY-NY)
/(LX-NX) : REM CALCULATE GRADIENT
14010 FOR X=INT(LX+.5) TO INT(NX+.5) STE
P SGN(NX-LX) : REM X FIXED
14020 Y=INT((X-LX)*S+LY+.5) : REM Y DERI
VED
14030 BI=CR*(4^FNBI(X)) : REM BITS TO MO
DIFY
14040 BM=3*(4^FNBI(X)) : REM BITS TO MAS
K
14050 P=FNCO(0) : REM THIS LOCATION
14060 IF P > 8191 AND P < 16192 THEN POK
E P,(PEEK(P) AND (255-BM)) OR BI
14070 REM WITH THE ABOVE VALUE
14080 NEXT X
14090 RETURN
```

```
10REM
20REM ------------------------------------
30REM
40REM MULTI-COLOUR HIRES : Y FIXED
50REM
60REM ------------------------------------
70REM
15000 S=0 : IF LY-NY <> 0 THEN S=(LX-NX)
/(LY-NY)
15010 FOR Y=INT(LY+.5) TO INT(NY+.5) STE
P SGN(NY-LY)
15020 X=INT(Y-LY)*S+LX+.5
15030 BI=CR*(4^FNBI(X))
15040 BM=3*(4^FNBI(X))
15050 P=FNCO(0)
15060 IF P > 8191 AND P < 16192 THEN POK
E P,(PEEK(P) AND (255-BM)) OR BI
15080 NEXT Y
15090 RETURN
```

FNBI(Z) effectively takes the bit position, as would be calculated in the old manner, and then divides by 2 and INTegerizes. FNBI now locates a bit pair (from 0 to 3) rather than an individual bit.

The routine at 14000 is that for drawing a straight line, when we take the X axis as fixed and derive Y values from there. This is the same as the equivalent two-colour high resolution routine, until we get to line 14030. At this point, it becomes rather more apparent that we are not merely switching individual bytes.

Take the bit pattern 00110001. This bit pattern is made up of four bit pairs. Bit pair 3 is 00, bit pair 2 is 11, bit pair 1 is 00, and bit pair 0 is 01. Suppose we want to change bit pair 0, over-writing it with 10. If we — as we would previously — OR with 2 (ie 00000010), then we produce a bit pattern 00110011. We do not produce 00110010, which is the desired effect.

What has happened is that 01 OR 10 is 11. To change the bits to the pattern we wish (ie 10) we have to 01 AND 00 OR 10 which is then 10.

First, therefore, we have to AND some zeros into the correct bit pair. We perform this zeroing by using a bit mask, which we call BM. BM is set equal to the value corresponding to a binary number with two bits (each equal to 1) at the correct bit pair position.

For example, if we consider the third bit pair (bits 4 and 5), 00110000, this is equal to 48(3*4^2). The binary we would like to use to AND is 11001111 (or 207), and this we find by subtracting 48 from 255.

The binary number is set within the pair by CR*4^(BITPAIR), where CR is the ColouR code to be used (the binary number is called BI).

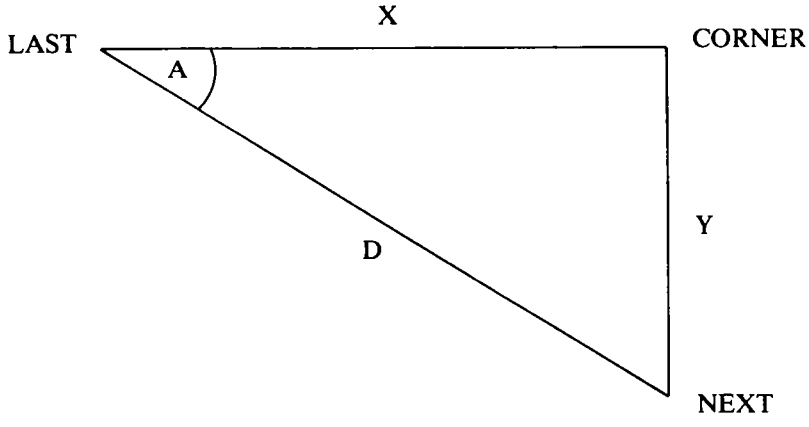
The next major difference from the two-colour version occurs, therefore, at line 14060. The value POKEd is slightly more complex in appearance because the value has to be first masked to zero. Apart from that there is little difference.

The multicolour system is easy to use, and there is no reason why (given patience, as it is so slow) complex effects cannot be produced.

APPENDIX J
Triangle Geometry

Triangle geometry is another name for trigonometry. We have been using trigonometry when we used the gradient (which we called S) to calculate coordinate values. Trigonometry is used in the turtle graphics routines to calculate new coordinates and new angles.

Figure J.1



The gradient is technically called the tangent, and to refer to the triangle shown in **Figure J.1**, we can see that the tangent of the angle we have labelled A, is Y/X. We show this by

TAN(A) = Y/X

There is another side to the triangle, the distance between the LAST point and the NEW point. We call this distance D. There are two trigonometrical ratios which use the (longest) side D

SIN(A) = Y/D
COS(A) = X/D

— you will realize that SIN(A) = COS(B) (or will you?)

If we know the value of the tangent (as we do with the gradient), we can use it to find the angle that corresponds to that tangent. We have a special way of naming the angle which corresponds to a specified tangent: we find the angle by use of the arctangent, ie

```
A = ATN(Y/X)
```

This is all you need to know to appreciate how the turtle routines actually work: a study of trigonometry is — of course — worthwhile in itself.

Conventionally, with reference to angle A, side D is the hypotenuse, Y the opposite, and Y the adjacent.

The main part of this book works on the development of a high resolution turtle graphics system for the Commodore 64. The discussion in the text is complemented with numerous appendices explaining specific topics not readily available to the CBM64 user.

The topics covered include PEEKS & POKES, binary and other arithmetic, the use of logical operators to produce special effects, the VIC-ROMAN II and CIA chips, the arrangement and relocation of memory, the theory of plotting, simple trigonometry.

Boris Allan only assumes knowledge of the Commodore 64 User Manual. All the moves, from relocating BASIC to the more advanced turtle graphics routines, are explained and documented in great detail.

Boris Allan is a well known contributor to a wide range of computer magazines and has written several highly successful books on home computing. He also writes a weekly commentary on the home computer market in Popular Computing Weekly.

ISBN 0 946408 15 7

£5.95 net